Innovative Integration

... real time solutions!

Sbc6713e User's Manual

Sbc6713e User's Manual

The Sbc6713e User's Manual was prepared by the technical staff of Innovative Integration on February 5, 2009.

For further assistance contact:

Innovative Integration
2390-A Ward Ave
Simi Valley, California 93065

PH: (805) 578-4260
FAX: (805) 578-4225

email: techsprt@innovative-dsp.com
Website: www.innovative-dsp.com

This document is copyright 2009 by Innovative Integration. All rights are reserved.

VSS \ Distributions \ Sbc6713e \ Documentation \ Manual \ Sbc6713eMaster.odm

#XXXXXX

Rev 1.0

Table of Contents

Introduction....11

Real Time Solutions!....11
Vocabulary....11
What is the SBC6713e? 11
What is C++ Builder?....12
What is Microsoft MSVC?....12
What kinds of applications are possible with Innovative Integration hardware?....12
Why do I need to use Malibu with my Baseboard?....12
Finding detailed information on Malibu....12
Online Help....13
Innovative Integration Technical Support....13
Innovative Integration Web Site....13
Typographic Conventions....13

Windows Installation....15

Host Hardware Requirements....15
Software Installation....15
Starting the Installation 16
The Installer Program....17
Tools Registration....19
Bus Master Memory Reservation Applet....19
Hardware Installation....20
After Power-up....21

JTAG Hardware Installation....22

JTAG Emulator Hardware Installation (for DSP boards Only)....22
PCI Pod-Based Emulator Installation....22
Baseboard Installation....23
A Few Considerations BEFORE Power-up....23
It cannot be overemphasized: 23
After Power-up....24
Code Composer Studio Setup with II Jtag....24
Setting up for a single processor with Spectrum Digital USB Jtag....28
Setting up for Multi Processors with Spectrum Digital USB Jtag....31
Borland Builder Setup and Use....34
Automatic saving of project files and forms during debugging....34
Static-binding of built executables. 35
Appropriate library and include paths. 35

SBC6713e Product Migration....37

Single Board Computer Hardware Features....37
Changes to SDRAM initialization. 37
Changes to Phase Locked Loop initialization....38
Changes to Application Programs....38
Upgrading the Firmware....38
(1) Changes to Loader program. 38
(2) Changes to the Coprocessor Talker(a.k.a Client)....39

(3) Changes to Flash Information Section. 39
(4) Changes to the Firmware....40

About the Baseboard....41

Single Board Computer Hardware Features....41
Digital Signal Processor....41
DSP External Memory....41
DSP Initialization....42
DSP JTAG Debugger Support....42
The Pismo Class Library....42
Simple To Use....43
Not Just for C++ Experts 44
Analog I/O Streams....44
Stream Objects and Device Drivers....45
Hardware Isolation and Independence....45
Stream I/O Types....45
Stream Buffer Model....46
Stream Internals....47
Multitasking Friendly....47
Using Analog Streams in an Application 47
Selecting the Stream Object....48
Selecting and Configuring Hardware....49
Selecting and Configuring Clocks....49
Selecting and Configuring Triggers....50
Selecting Pretriggering Modes....50
Interrupt Handling....52
Interrupts in a C++ Environment....52
The Pismo Solution....53
Class Irq....54
Interrupt Lock Classes....54
Interrupt Binder Templates 54
Class InterruptHandler....54
Class ClassMemberHandler Template....54
Class FunctionHandler Template....55
EDMA and QDMA Handling....56
Class DmaSettings....56
Class Qdma....56
Class Edma....58
Linked and Chained blocks....60
Class EdmaMaster....60
SBC6713e Example Programs....61
The Next Step: Developing Custom Code....62

Communication with the Host....63

Overview....63
Packetized Message Interface....63
Communication between Host PC and DSP Co-Processor....64
Communication between Host PC and DSP Target Processor....66
C++ Terminal I/O....69
Target Software....69
Tutorial....69

Detailed Host-Target Communication....71
Overview....71
I. COFF file download....71
II. Sending data to Target....72
*** All Data Packets intended to reach the Target C6713 must have an even size of 32-bit words and a minimum of 4 words....74
II. Receiving data from Target....74
III. Sending RESET command to Target....76
IV. Requesting Board information....77
V. Write Board information....77
VI. Burning Firmware....78
Building a Target DSP Project....81
Writing a Program....86
Host Tools for Target Application Development....86
Components of Target Code (.cpp, .tcf, .cmd, .pjt)....86
Edit-Compile-Test Cycle using Code Composer Studio....87
Automatic projectfile creation....87
Rebuilding a Project....87
IIMain replaces main....87
Running the Target Executable....87
Note:....88
Anatomy of a Target Program....88
Use of Library Code....89
Example Programs....89
The Next Step: Developing Custom Code....90
Developing Host Applications....15
Borland Turbo C++....15
Microsoft Visual Studio 2005....17
Applets....19
Common Applets....20
Registration Utility (NewUser.exe)....20
Reserve Memory Applet (ReserveMemDsp.exe)....22
Data Analysis Applets....23
Binary File Viewer Utility (BinView.exe)....23
Target Programming Applets....24
Target Project Copy Utility (CopyCcsProject.exe)....24
Demangle Utility (Demangle.exe)....24
COFF Section Dump Utility (CoffDump.exe)....24
JTAG Diagnostic Utility (JtagDiag.exe)....25
RtdxTerminal - Terminal Emulator....25
Important Note:....25
Terminal Emulator Menu Commands....25
The File Menu....26
The DSP Menu....26
The Form Menu....27
The Help Menu....27
Options Tab:....27

Display Group....28
Sounds Group....29
Coff Load Group....29
Debugger Group....29
Terminal Emulator Command Line Switches....29
Applets for SBC6713e Baseboard....30
Board Finder Utility (Sbc6713eFinder.exe)....30
Target Project Copy Utility (CopyCcsProject.exe)....31
Flash Conversion Utility (PromImage.exe)....32
FlashBurn 32
Usage....33
Custom Logic Development....37
Overview....37
Target Devices....37
Code Development Tools....38
Steps to Successful Use....38
Using FPGA Framework....38
UART Overview (RS232 Asynchronous Serial Port)....40
Developing FPGA Firmware....40
Logic Files and Hierarchy....41
Fitting Results for the Framework Logic....42
Adding functionality to the Framework Logic....44
Logic Design Methods and Code Development....44
DSP Interface Logic....44
DSP EMIF Data Read Interface....47
DSP EMIF Write Interface....49
Digital IO from the Spartan IIE....51
Clock Domains....51
Constraints....52
Timing Constraints....52
IO Standard Constraints....53
Pin Constraints....53
Pin Assignments For Spartan IIE (300K-600K)....53
Simulation....161
Required Software and Hardware....161
Setting Up the Simulation....161
Loading the Testbench....161
Running the Simulations....161
Modifying the Simulations....162
Omnibus Interface....164
Burst Reads And Writes....165
Loading the Logic Image....168
FlashBurn Utility....164
Debugging Custom Logic....165
Built-in Test Modes....165
Xilinx ChipScope 166
Declaration Of ChipScope Core in VHDL....168
Creating a Binary File....171
baseboard31, baseboard32 and baseboard54 targets....171

baseboard67 and baseboard6711 Targets....172
The Flash Burn Utility....172
Target Page....172
Talker Page....173
JTAG Download....174
Flash ROM Download....175
Downloading the Flash Support Code....175
Flash Page....175
Controlling Region of Erasure....175
Re-burning the Talker Image....176
Warning:....176
Burning a User-Written Application....176
Info Page....176
Example Burn Sequence....177
baseboard32:....178
baseboard6x:....178
baseboard6711:....178
Common Problems when Embedding Code....178
OLED Display Adapter....180
Description....180
Button Interface....181
Display Interface....182
Serial Word Format....182
Data/Command Pacing....183
Required steps to use the display....184
Appendices....185
Connector pinouts....185
JP4, JP8 - OMNIBUS I/O Connectors....185
JP5- OMNIBUS I/O Connectors....185
JP6, 7, 9, 10 – OMNIBUS Bus Connectors....189
JP1 – Digital I/O Connector....191
JH1 – FPDP Transmit Port Connector....192
JH2 – FPDP Receive Port Connector....194
JP2 - SyncLink/ClkLink....196
JP13, JP15 – Processor Serial Port Connectors....197
JP23 – JTAG Debugger Connector (Rev C)....198
JP22 – Power Input Connector....199
JP3 – Asynchronous Serial Port Connector....200
JP12 - Power Test Connector (Rev C)....200
JP25 - Xilinx JTAG Connector (i.e. XC2Sx00E-FG456) (Rev C)....201
JP21 - Xilinx JTAG Connector (i.e. XC2S50E-TQ144) (Rev C)....202
Board Layout (Rev B)....203
Board Layout (Rev C)....204

List of Tables

Table 1. Production Hardware Changes - Rev B vs C....37
Table 2. SDRAM Register Changes....37
Table 3. PLL Register Changes....38
Table 4. ‘C6713 DSP EMIF Control Register Initialization Values....42
Table 5. Device Driver and Stream Classes....45
Table 6. Stream object Clock Methods....49
Table 7. Stream object Pretrigger Methods....50
Table 8. Stream object Start Trigger Methods....50
Table 9. Stream object Stop Trigger Methods....51
Table 10. Stream object Retrigger Methods....51
Table 11. Interrupt Lock Classes....54
Table 12. SBC6713e Example Programs....61
Table 13. Pismo Example Programs....90
Table 14. I/O Standard - LVTTL....53
Table 15. Common Problems when Embedding Code in Flash ROM....178
Table 16. OMNIBUS Bus Connectors....189
Table 17. I/O Module Bus Connectors....190
Table 18. Digital I/O Connector....191
Table 19. SyncLink Connector....196
Table 20. Processor Serial Port Connector....197
Table 21. JTAG Debugger Connector....198
Table 22. Power Input Connector....199
Table 23. Asynchronous Serial Port Connector....200
Table 24. Xilinx JTAG Connector for XC2S600E-FG456....201
Table 25. Xilinx JTAG Connector for XC2S50E-TQ144....202

List of Figures

Figure 1. Vista Verificaton Dialog....16
Figure 2. Innovative Install Program....17
Figure 3. Progress is shown for each section....18
Figure 4. ToolSet registration form....19
Figure 5. BusMaster configuration....19
Figure 6. Installation complete....20
Figure 7. RtdxTerminal Options....28
Figure 8. FPGA Framework Overview....38
Figure 9. Framework Logic Block Diagram....39
Figure 10. UART Block Diagram....40
Figure 11. Logic Files and Hierarchy....42
Figure 12. Map Results....43
Figure 13. PAR Results....44
Figure 14. DSP Asynchronous Read Timing (Courtesy of Texas Instrument)....45
Figure 15. DSP Asynchronous Write Timing (Courtesy of Texas Instrument)....45
Figure 16. DSP SBSRAM Read Timing (Courtesy of Texas Instruments)....46
Figure 17. DSP SBSRAM Write Timing (Courtesy of Texas Instruments)....47
Figure 18. EMIF B Data Bus Read Interface Diagram....48
Figure 19. Data Mux....49
Figure 20. Incoming DSP Data....50
Figure 21. DSP Controls Signals....50
Figure 22. FIFO Component Instantiation....50
Figure 23. Using the FIFO Interrupt Filter....51
Figure 24. DSP Burst Read and Writes....51
Figure 25. BUFGDLL Schematic (Courtesy of Xilinx)....52
Figure 26. Clock Period Constraints....52
Figure 27. DSP Signals Timing Constraints....53
Figure 28. IO Standard Constraint....53
Figure 29. VHDL Procedure for Simulating a DSP Async Access....163
Figure 30. VHDL Testbench Code For Omnibus....163
Figure 31. Omnibus Writes And Reads....164
Figure 32. VHDL Procedure For DSP Burst Read Access....166
Figure 33. Data Read From FPDP Tx FIFO And Data Write to FPDP Rx FIFO....166
Figure 34. DSP Burst Reads From FPDP Rx FIFO....167
Figure 35. Burst Writes And Reads....167
Figure 36. Sample For Simulating DSP Interrupt Servicing....167
Figure 37. Click on PROM File Option....168
Figure 38. PROM Configuration....161
Figure 39. Select “Auto Select PROM”....162
Figure 40. Select the BIT file....163
Figure 41. PROM File Generation....164
Figure 42. Flash Image Burner....165
Figure 43. Chip Scope Big Picture....166
Figure 44. Xilinx Parallel Cable IV (Top View) (Courtesy of Xilinx)....167
Figure 45. High Performance Ribbon Cable (Courtesy Of Xilinx)....167
Figure 46. Xilinx Cable Interface Connector (Courtesy Of Xilinx)....167
Figure 47. SBC6713e - JP25, Spartan IIe(300K-600K) JTAG Connector....167

Figure 48. Chip Scope Core Declarations....168
Figure 49. Chip Scope Core Instantiation and Use....169
Figure 50. Chip Scope Analyzer Screen Shot....169
Figure 51. FPDP JH1 Tx Port Connector....193
Figure 52. FPDP JH2 Rx Port Connector....195
Figure 53. JP2 SyncLink Connector Pin Orientation....196
Figure 54. JP13, JP15 DSP Serial Port Connector....197
Figure 55. JP23 JTAG Debugger Connector....198
Figure 56. Power Connector Pin Positions (side view, from front of connector, showing connector keying and locking tab along with printed circuit board position)....199
Figure 57. Mechanical Drawing (Board Revision C)....204

Introduction

Real Time Solutions!

Thank you for choosing Innovative Integration, we appreciate your business! Since 1988, Innovative Integration has grown to become one of the world's leading suppliers of DSP and data acquisition solutions. Innovative offers a product portfolio unrivaled in its depth and its range of performance and I/O capabilities .

Whether you are seeking a simple DSP development platform or a complex, multiprocessor, multichannel data acquisition system, Innovative Integration has the solution. To enhance your productivity, our hardware products are supported by comprehensive software libraries and device drivers providing optimal performance and maximum portability.

Innovative Integration's products employ the latest digital signal processor technology thereby providing you the competitive edge so critical in today's global markets. Using our powerful data acquisition and DSP products allows you to incorporate leading-edge technology into your system without the risk normally associated with advanced product development. Your efforts are channeled into the area you know best ... your application.

Vocabulary

What is the SBC6713e?
The SBC6713e is Innovative Integration's stand-alone baseboard architecture that integrates modularized, high-performance analog and digital peripherals with two high-performance DSP and peripheral cores. The SBC6713e is a derivative of the SBC6711 (Pantera), having an Ethernet port instead of a USB port. The more common and flexible Ethernet port provides greater throughput (approaching 100Mb/s) compared to the slower USB port.

The baseboard includes an onboard TMS320C6713 DSP with 32 MB cached SDRAM. The DSP accesses to the entire baseboard peripheral complement directly as memory-mapped devices. The baseboard supports speeds approaching 100 Mbit/sec, a packet-message-based Ethernet bus interface, two variable-function Omnibus I/O module sites, dedicated Front Panel Data Port (FPDP) and SyncLink buses for inter-board connectivity, a precision DDS timebase to serve as an accurate, programmable clock source from DC-25 MHz and a programmable digital I/O port. Because of the onboard DSP, the baseboard is capable of performing data collection, servo or other real-time processing and data movement automatically without Host PC CPU involvement.

What is C++ Builder?
C++ Builder is a general-purpose code-authoring environment suitable for development of Windows applications of any type. Armada extends the Builder IDE through the addition of functional blocks (VCL components) specifically tailored to perform real-time data streaming functions.

What is Microsoft MSVC?
MSVC is a general-purpose code-authoring environment suitable for development of Windows applications of any type. Armada extends the MSVC IDE through the addition of dynamically created MSVC-compatible C++ classes specifically tailored to perform real-time data streaming functions.

What kinds of applications are possible with Innovative Integration hardware?
Data acquisition, data logging, stimulus-response and signal processing jobs are easily solved with Innovative Integration baseboards using the Malibu software. There are a wide selection of peripheral devices available in the Matador DSP product family, for all types of signals from DC to RF frequency applications, video or audio processing. Additionally, multiple Innovative Integration baseboards can be used for a large channel or mixed requirement systems and data acquisition cards from Innovative can be integrated with Innovative's other DSP or data acquisition baseboards for high-performance signal processing.

Why do I need to use Malibu with my Baseboard?
One of the biggest issues in using the personal computer for data collection, control, and communications applications is the relatively poor real-time performance associated with the system. Despite the high computational power of the PC, it cannot reliably respond to real-time events at rates much faster than a few hundred hertz. The PC is really best at processing data, not collecting it. In fact, most modern operating systems like Windows are simply not focused on real-time performance, but rather on ease of use and convenience. Word processing and spreadsheets are simply not high-performance real-time tasks.

The solution to this problem is to provide specialized hardware assistance responsible solely for real- time tasks. Much the same as a dedicated video subsystem is required for adequate display performance, dedicated hardware for real-time data collection and signal processing is needed. This is precisely the focus of our baseboards – a high performance, state-of-the-art, dedicated digital signal processor coupled with real-time data I/O capable of flowing data via a 64-bit PCI bus interface.

The hardware is really only half the story. The other half is the Malibu software tool set which uses state of the art software techniques to bring our baseboards to life in the Windows environment. These software tools allow you to create applications for your baseboard that encompass the whole job - from high speed data acquisition, to the user interface.

Finding detailed information on Malibu
Information on Malibu is available in a variety of forms:

- Data Sheet (http://www.innovative-dsp.com/products/malibu.htm)
- On-line Help

- Innovative Integration Technical Support
- Innovative Integration Web Site (www.innovative-dsp.com)

Online Help

Help for Malibu is provided in a single file, Malibu.chm which is installed in the Innovative\Documentation folder during the default installation. It provides detailed information about the components contained in Malibu - their Properties, Methods, Events, and usage examples. An equivalent version of this help file in HTML help format is also available online at http://www.innovative-dsp.com/support/onlinehelp/Malibu.

Innovative Integration Technical Support

Innovative includes a variety of technical support facilities as part of the Malibu toolset. Telephone hotline supported is available via

Hotline (805) 578-4260 8:00AM-5:00 PM PST.

Alternately, you may e-mail your technical questions at any time to:

techsprt@innovative-dsp.com.

Also, feel free to register and browse our product forums at http://forum.iidsp.com/, which are an excellent source of FAQs and information submitted by Innovative employees and customers.

Innovative Integration Web Site

Additional information on Innovative Integration hardware and the Malibu Toolset is available via the Innovative Integration website at www.innovative-dsp.com

Typographic Conventions

This manual uses the typefaces described below to indicate special text.

Typeface	Meaning
`Source Listing`	Text in this style represents text as it appears onscreen or in code. It also represents anything you must type.
Boldface	Text in this style is used to strongly emphasize certain words.
Emphasis	Text in this style is used to emphasize certain words, such as new terms.

Cpp Variable	Text in this style represents C++ variables
Cpp Symbol	Text in this style represents C++ identifiers, such as class, function, or type names.
KEYCAPS	Text in this style indicates a key on your keyboard. For example, “Press ESC to exit a menu”.
Menu Command	Text in this style represents menu commands. For example “Click View | Tools | Customize”

Windows Installation

This chapter describes the software and hardware installation procedure for the Windows platform (WindowsXP and Vista).

> **Do *NOT*** install the hardware card into your system at this time. This will follow the software installation.

Host Hardware Requirements

The software development tools require an IBM or 100% compatible Pentium IV - class or higher machine for proper operation. An Intel-brand processor CPU is *strongly recommended*, since AMD and other "clone" processors are not guaranteed to be compatible with the Intel MMX and SIMD instruction-set extensions which the Armada and Malibu Host libraries utilize extensively to improve processing performance within a number of its components. The host system must have at least 128 Mbytes of memory (256MB recommended), 100 Mbytes available hard disk space, and a DVD-ROM drive. Windows2000 or WindowsXP (referred to herein simply as *Windows*) is required to run the developer's package software, and are the target operating systems for which host software development is supported.

Software Installation

The development package installation program will guide you through the installation process.

> **Note:** Before installing the host development libraries (VCL components or MFC classes), you must have Microsoft MSVC Studio (version 9 or later) and/or Codegear RAD Studio C++ (version 11) installed on your system, depending on which of these IDEs you plan to use for Host development. If you are planning on using these environments, it is imperative that they are tested and known-operational before proceeding with the library installation. If these items are not installed prior to running the Innovative Integration install, the installation program will not permit installation of the associated development libraries. However, drivers and DLLs may be installed to facilitate field deployment.

You must have **Administrator Privileges** to install and run the software/hardware onto your system, refer to the Windows documentation for details on how to get these privileges.

Starting the Installation

To begin the installation, start Windows. Shut down all running programs and disable anti-virus software. Insert the installation **DVD**. If Autostart is enabled on your system, the install program will launch. If the DVD does not Autostart, click on Start | Run... Enter the path to the **Setup.bat** program located at the root of your DVD-ROM drive (i.e. **E:\Setup.bat**) and click "OK" to launch the setup program.

> **SETUP.BAT** detects if the OS is 64-bit or 32-bit and runs the appropriate installation for each environment. It is important that this script be run to launch an install.

When installing on a Vista OS, the dialog below may pop up. In each case, select "Install this driver software anyway" to continue.

Windows Security
Windows can't verify the publisher of this driver software
Don't install this driver software
You should check your manufacturer's website for updated driver software for your device.
Install this driver software anyway
Only install driver software obtained from your manufacturer's website or disc. Unsigned software from other sources may harm your computer or steal information.
See details

Figure 1. Vista Verificaton Dialog

The Installer Program

After launching Setup, you will be presented with the following screen.

Figure 2. Innovative Install Program

Using this interface, specify which product to install, and where on your system to install it.

1) Select the appropriate product from the Product Menu.

2) Specify the path where the development package files are to be installed. You may type a path or click "**Change**" to browse for, or create, a directory. If left unchanged, the install will use the default location of "C:\Innovative".

3) Typically, most users will perform a "Full Install" by leaving all items in the "**Components to Install**" box checked. If you do not wish to install a particular item, simply uncheck it. The Installer will alert you and automatically uncheck any item that requires a development environment that is not detected on your system.

4) Click the Install button to begin the installation.

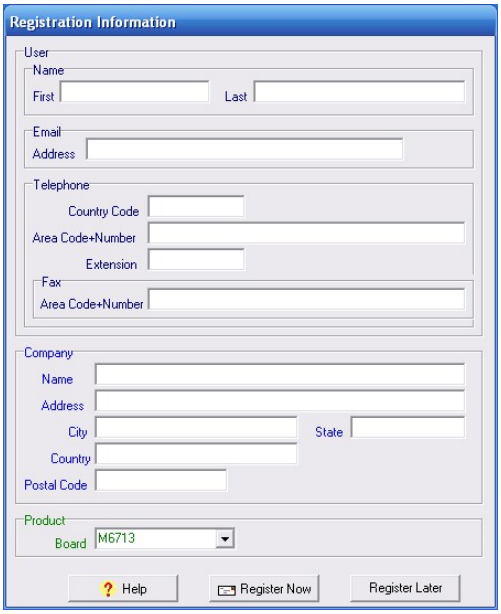
> **Note:** The default "Product Filter" setting for the installer interface is "Current Only" as indicated by the combo box located at the top right of the screen. If the install that you require does not appear in the "Product Selection Box" (1), Change the "Product Filter" to "Current plus Legacy".

Each item of the checklist in the screen shown above, has a sub-install associated with it and will open a sub-install screen if checked. For example, the first sub-install for “Quadia - Applets, Examples, Docs, and Pismo libraries” is shown below.

The installation will display a progress window, similar to the one shown below, for each item checked.

Figure 3. Progress is shown for each section.

Tools Registration



At the end of the installation process you will be prompted to register. If you decide that you would like to register at a later time, click "Register Later".

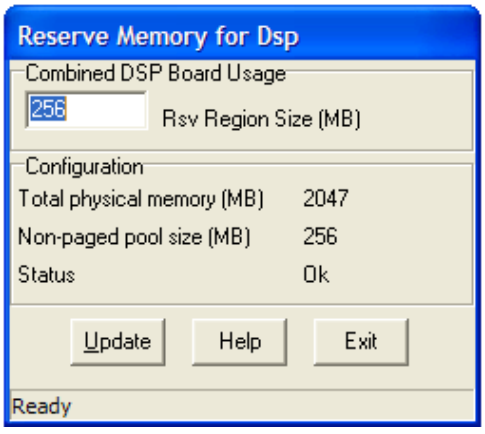
When you are ready to register, click Start | All Programs | Innovative | <Board Name> | Applets. Open the New User folder and launch **NewUser.exe** to start the registration application. The registration form to the left will be displayed.

Before beginning DSP and Host software development, you must register your installation with Innovative Integration. Technical support will not be provided until registration is successfully completed. Additionally, some development applets will not operate until unlocked with a passcode provided during the registration process.

It is recommend that you completely fill out this form and return it to Innovative Integration, via email or fax. Upon receipt, Innovative Integration will provide access codes to enable technical support and unrestricted access to applets.

Figure 4. ToolSet registration form

Bus Master Memory Reservation Applet.



At the conclusion of the installation process, **ReserveMem.exe** will run (except for SBC products). This will allow you to set the memory size needed for the busmastering to occur properly. This applet may be run from the start menu later if you need to change the parameters.

For optimum performance each Matador Family Baseboard requires 2 MB of memory to be reserved for its use. To reserve this memory, the registry must be updated using the ReserveMem applet. Simply select the **Number of Baseboards** you have on your system, click **Update** and the applet will update the registry for you. If at any time you change the number of boards in your system, then you must invoke this applet found in Start | All Programs | Innovative | <target board> | Applets | Reserve Memory.

After updating the system exit the applet by clicking the **exit** button to resume the installation process.

Figure 5. BusMaster configuration

At the end of the install process, the following screen will appear.

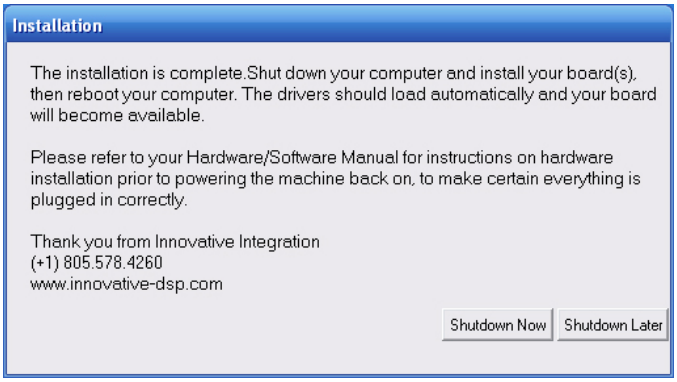

Installation
The installation is complete.Shut down your computer and install your board(s), then reboot your computer. The drivers should load automatically and your board will become available.
Please refer to your Hardware/Software Manual for instructions on hardware installation prior to powering the machine back on, to make certain everything is plugged in correctly.
Thank you from Innovative Integration
(+1) 805.578.4260
www.innovative-dsp.com
Shutdown Now
Shutdown Later

Figure 6. Installation complete

Click the "Shutdown Now" button to shut down your computer. Once the shutdown process is complete unplug the system power cord from the power outlet and proceed to the next section, "Hardware Installation."

Hardware Installation

Now that the software components of the Development Package have been installed the next step is to configure and install your hardware. Detailed instructions on board installation are given in the Hardware Installation chapter, following this chapter.

> **IMPORTANT:** Many of our high speed cards, especially the PMC and XMC Families, require forced air from a fan on the board for cooling. Operating the board without proper airflow may lead to improper functioning, poor results, and even permanent physical damage to the board. These boards also have temperature monitoring features to check the operating temperature. The board may also be designed to intentionally fail on over-temperature to avoid permanent damage. See the specific hardware information for airflow requirements.

After Power-up

After completing the installation, boot your system into Windows.

Innovative Integration boards are plug and play compliant, allowing Windows to detect them and auto-configure at start-up. Under rare circumstances, Windows will fail to auto-install the device-drivers for the JTAG and baseboards. If this happens, please refer to the "TroubleShooting" section.

JTAG Hardware Installation

JTAG Emulator Hardware Installation (for DSP boards Only)

First, the emulator hardware must be configured and installed into your PC. The emulator hardware is described in the table below:

Type	Features
Pod-based	Uses a special ribbon cable with integrated line drivers to connect the target DSP emulation signals to the JTAG debugger card. Usable on 3.3 volt or 5 volt designs. (Including 'C54x and 'C6x)

PCI Pod-Based Emulator Installation

To install the PCI pod based emulator, follow the instructions below:

5) Perform the board installation in an "ESD" or static safe workstation employing a static-dissipative bench mat. Wear a properly grounded wrist strap or other personal anti static device. Stand on an anti static mat or a static-dissipative surface.

6) Shut down Windows, power-off the host system and unplug the power cord.

7) Touch the chassis of the host computer system to dissipate any static charge.

8) Remove the card from its protective static-safe shipping container, being careful to handle the card only by the edges.

9) Touch the chassis of the PC to dissipate any built up static charge.

10) Securely install the JTAG board in an available PCI slot in the host computer.

11) Connect the JTAG pod to the host-pod cable. Connect the host-pod cable to the connector located on the end bracket of the JTAG PCI plug-in board.

Baseboard Installation

To install the baseboard:

12) Perform the board installation in an “ESD” or static safe workstation employing a static-dissipative bench mat. Wear a properly grounded wrist strap or other personal anti static device. Stand on an anti static mat or a static-dissipative surface.

13) Shut down Windows and power-off the host system and unplug the power cord

14) Touch the chassis of the host computer system to dissipate any static charge.

15) Remove the card from its protective static-safe shipping container, being careful to handle the card only by the edges.

16) Touch the chassis of the PC to dissipate any built up static charge.

17) Connect the 14-pin connector on the JTAG PCI pod to the DSP board JTAG connector (Non-DSP board users skip this step).

18) Securely install the baseboard into an available PCI slot in the host computer.

IMPORTANT: Many of our high speed cards, especially the PMC and XMC Families, require forced air from a fan on the board for cooling. Operating the board without proper airflow may lead to improper functioning, poor results, and even permanent physical damage to the board. These boards also have temperature monitoring features to check the operating temperature. The board may also be designed to intentionally fail on over-temperature to avoid permanent damage. See the specific hardware information for airflow requirements.

A Few Considerations BEFORE Power-up

Double-check all connections before applying power. Ensure that the JTAG and baseboard cards seated correctly in the slot.

It cannot be overemphasized:
Double check your cabling **BEFORE** connecting to the baseboard. **DO NOT** hot plug the cables. Hot plugging cables can cause latch-up of components on the card and damage them irreparably. Be aware that the cables to analog inputs are an important part of keeping signals clean and noise-free. Shielded cables and differential inputs (where applicable) help to control and reduce noise.

After Power-up
After completing the installation, boot your system into Windows.

Innovative Integration boards are plug and play compliant, allowing Windows to detect them and auto-configure at start-up. Under rare circumstances, Windows will fail to auto-install the device-drivers for the JTAG and baseboards. If this happens, please refer to the "TroubleShooting" section.

The next section is NOT used with the Non-DSP products. If the board you are installing is a Non-DSP product, the installation is complete.

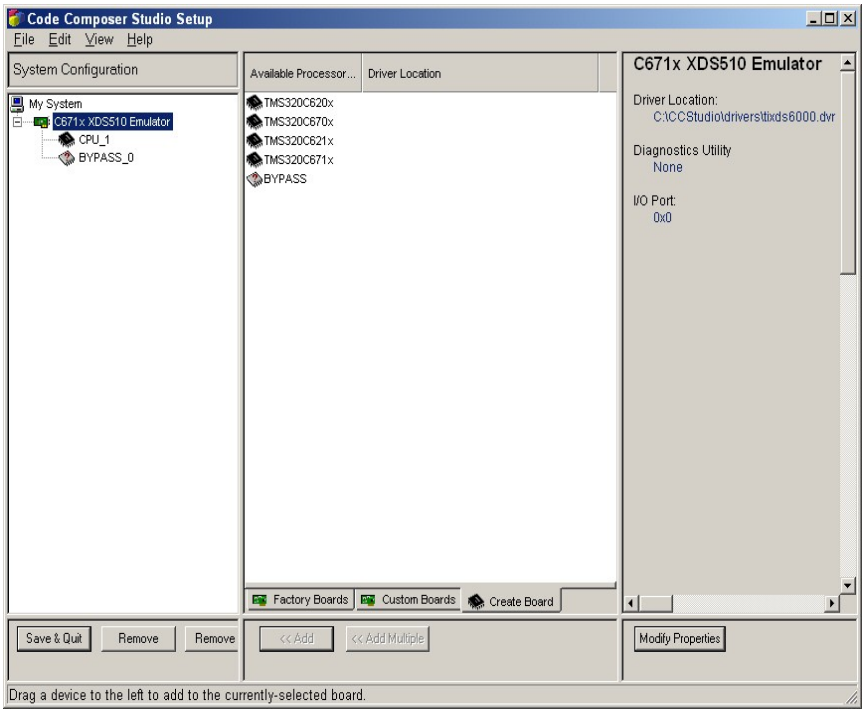
Code Composer Studio Setup with II Jtag

To setup Code Composer Studio and activate the Innovative Integration-supplied CodeHammer JTAG board driver, the Code Composer Studio Setup Utility must be run. Since the Code Hammer debugger is XDS510- compatible, Code Composer Studio setup must be configured to use the XDS510 driver for the C6xxx. To do this:

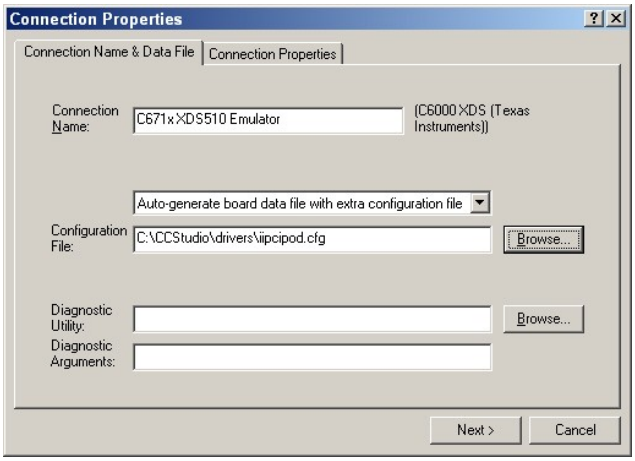
19) Launch the Code Composer Studio Setup Utility and remove the default simulator driver from the System Configuration. (right click the default simulator in the "System Configuration" pane and select "Remove")

20) Click the C6xxx XDS driver from the "Available Emulator Types" control within the setup utility and drag it into the "System Configuration" control.

21) Once your emulator is added, a list of Available Processors is presented. Add the appropriate processors for your board as shown in the example. The example shows a set-up that is configured for an SBC6713e baseboard. The C671x emulator is selected as the baseboard uses the TMS320C6713 DSP.

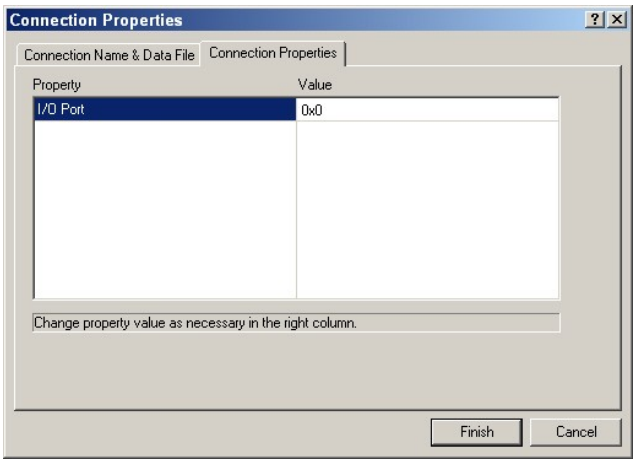


22) Right-click on the C6xxx XDS emulator in the “System Configuration” Pane and select “Properties…” to invoke the **Connection Properties** Dialog for the driver.



23) Under the “Connection Name & Data File” tab, the “**Connection Name**” edit box should match the emulator selected in the “**System Configuration**” Pane of the previous window. Change the “**Configuration File**” combo box to Auto-generate board data file with extra configuration file. Change the “**Configuration File**” edit box to "<drive>:\Cstudio\Drivers\IIPciPod.cfg". *<drive> is the letter for the drive onto which CCS is installed.*

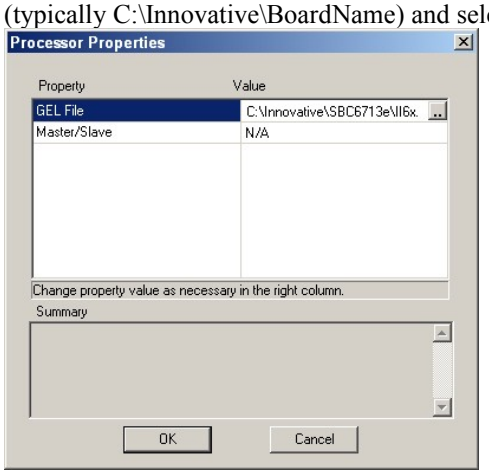
24) Click the “**Connection Properties**” tab. Set the I/O port value for the driver to virtual device address “0x0” and click “**Finish**”.



25) The main Code Composer Studio Setup window is now back in focus. The processor must now be configured. To do this: select the processor as shown in the “**System Configuration**” Pane (in our example “CPU_1”). Right click “CPU_1” and select “Properties…”.

26) The “**Processor Properties**” screen will be presented. Click GEL File, click the ellipsis (…) and navigate to the Innovative Integration board install directory

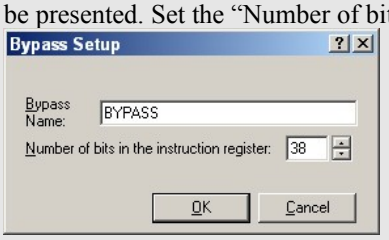
(typically C:\Innovative\BoardName) and select “II6x.gel”. Click “OK”.



27) Click “**Save & Quit**” to save the configuration and exit the setup tool. You will then be prompted to launch **Code Composer Studio**.

Note: For multi-target boards such as the Quadia, one processor should be added for each device in the JTAG scan path.

Note: The SBC6713e has (2) DSPs, a C6713 and a DM642. Typically the DM642 should be set to “BYPASS” by selecting “BYPASS” from the “Available Emulator Types” control within the setup utility and drag it into the “System Configuration” control. Once this is done, the following screen will be presented. Set the “Number of bits in the instruction register” to “38” and click “OK”



If you encounter difficulty launching CCS

28) Run the **JtagDiag.exe** utility (Start | All Programs | Innovative | Common Applets | JTAG Diagnostics) to reset the debugger interface.

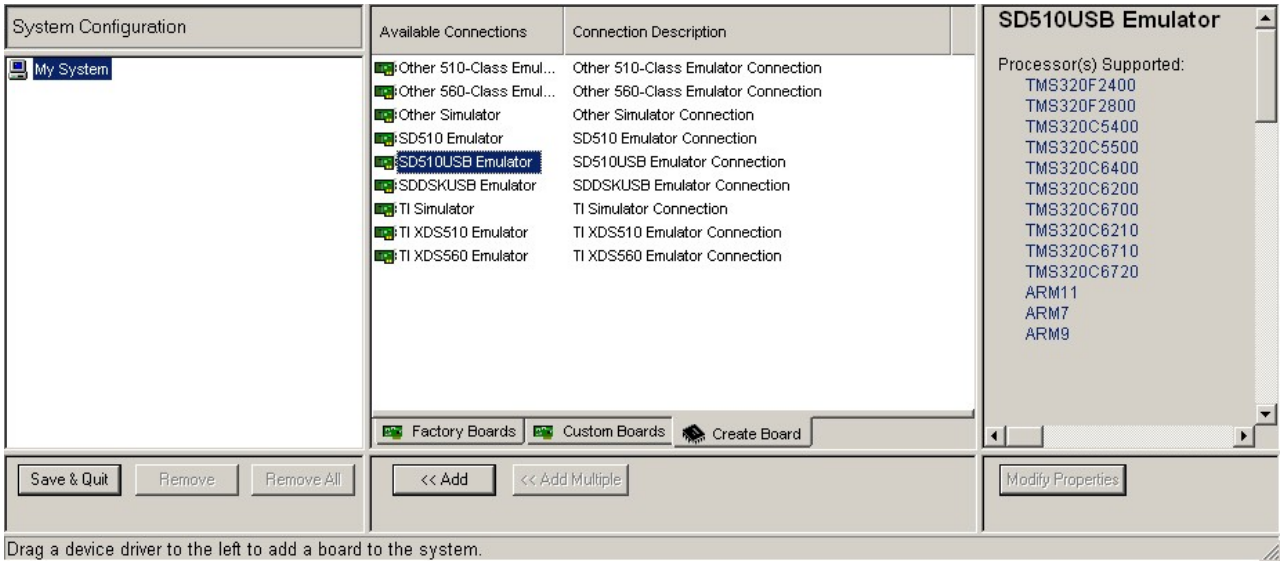
29) Run the board **Downloader utility** (Start | All Programs | Innovative | <Board Name> |<Applets> — Open the **Downloader Folder** and double click “**Downloader.exe**”) and press the Boot button (*Light Bulb icon*), to boot a default target application.

30) Restart Code Composer Studio.

Setting up for a single processor with Spectrum Digital USB Jtag.

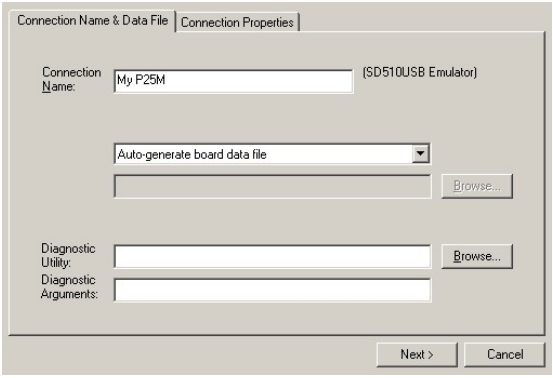
First remove any previous setups in the CCS Setup application.

Add one of the USB SD type driver.



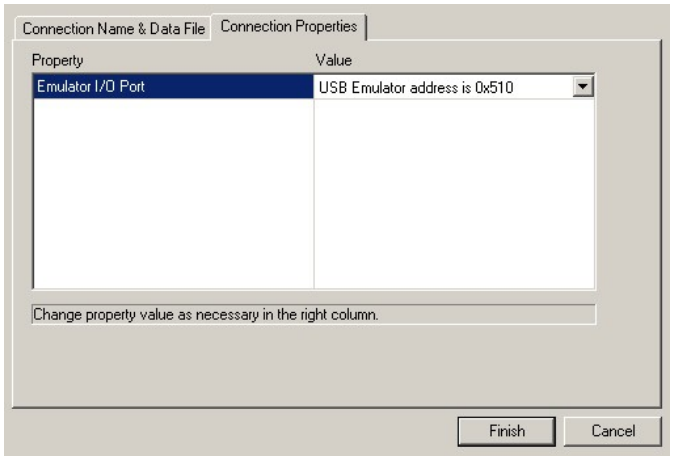
You will see the following screen.

Fill out the name of the board you are using, this can be any name you like.



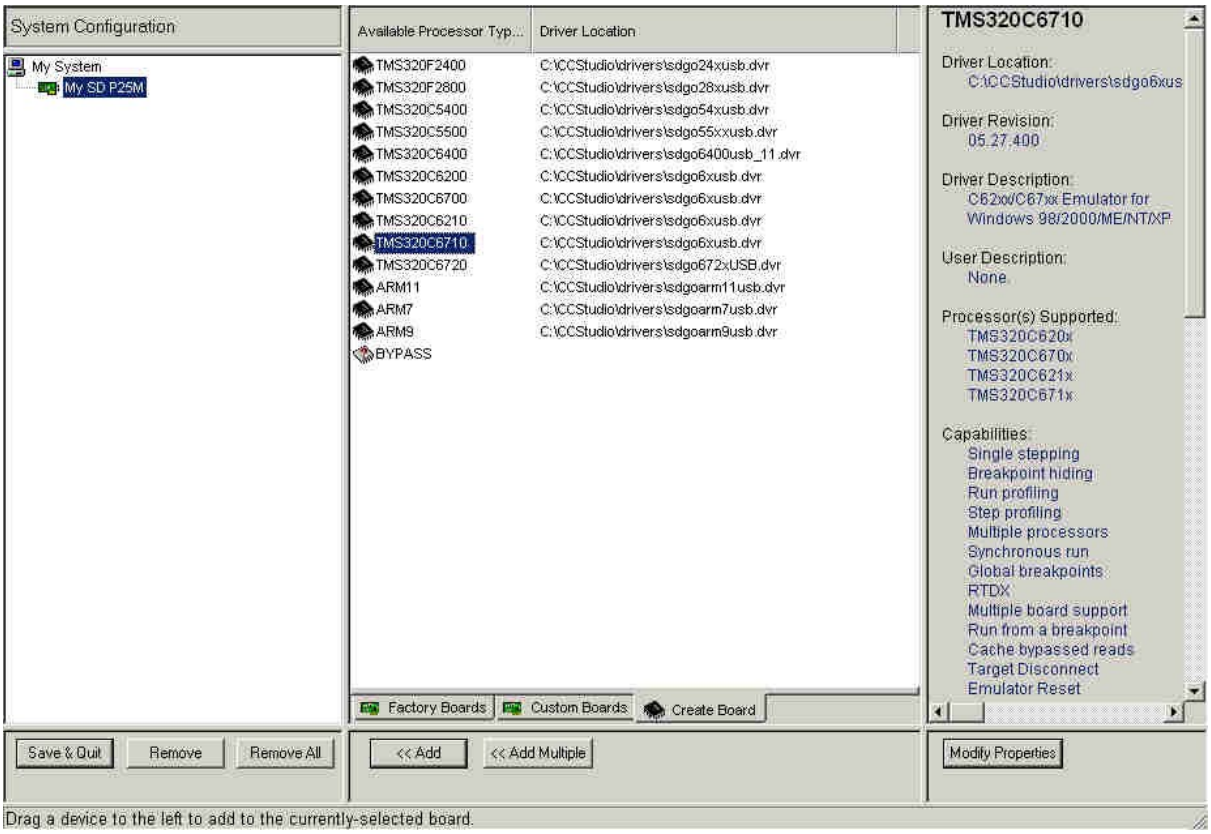
Hit next or move to the next tab

This address should match up with the address in the SdConfig.exe utility

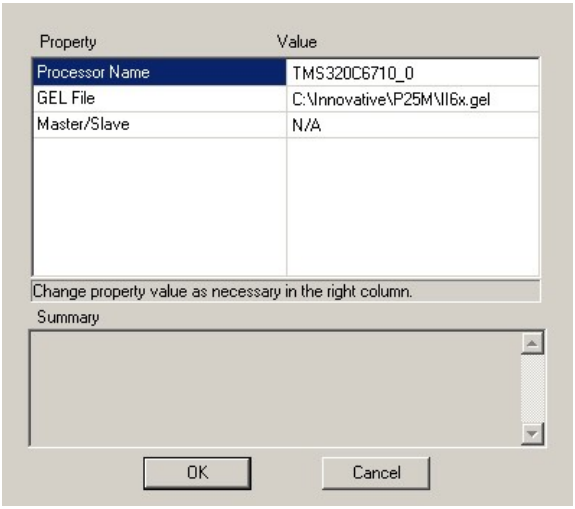


Now we add a processor

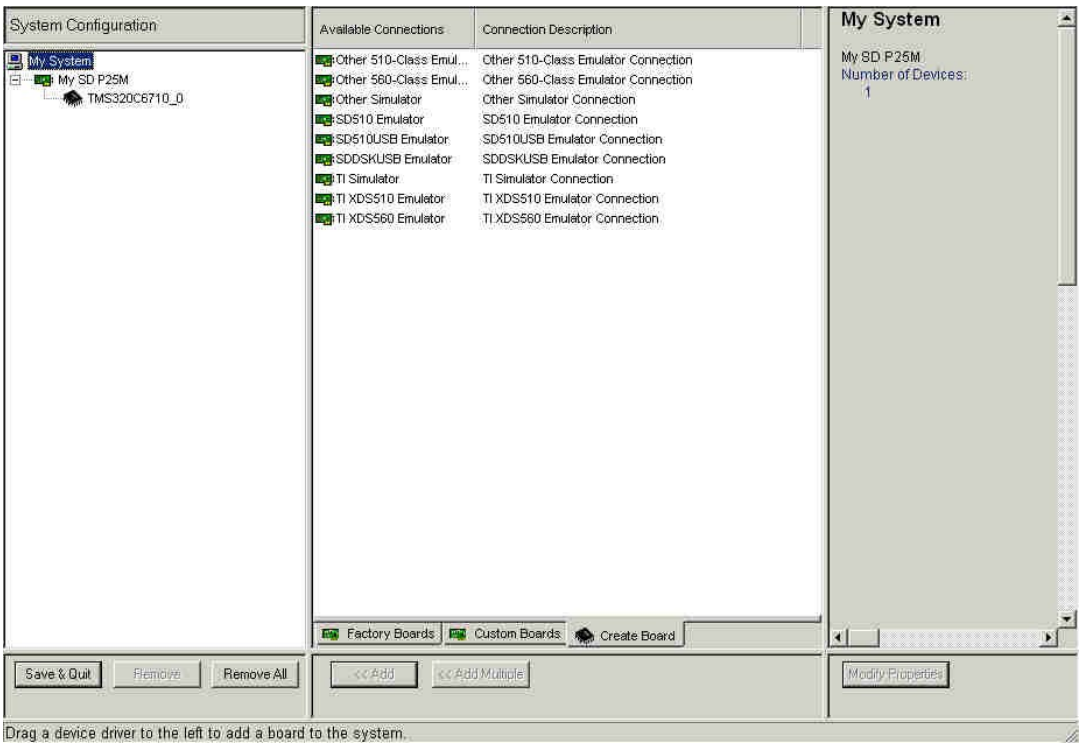
Each if the II boards have different processors so match up the closest one for your board.



Use the property sheet to find the Gel file from Innovative for your specific board.



Your system will look similar to this. Save the configuration and quit.



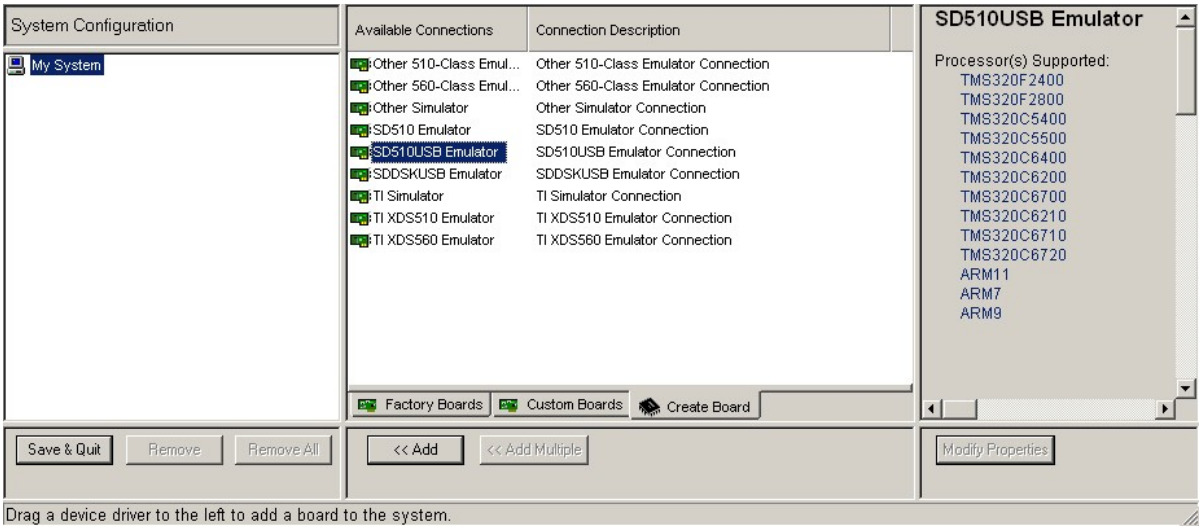
Setting up for Multi Processors with Spectrum Digital USB Jtag.

For the multi-processor setups use the following type setup. This includes the SBC6713e, Quadia, Q6x type Innovative boards.

The SBC6713e board shown will be similar in setup with the other boards. The differences will be in the types of processors and the number added.

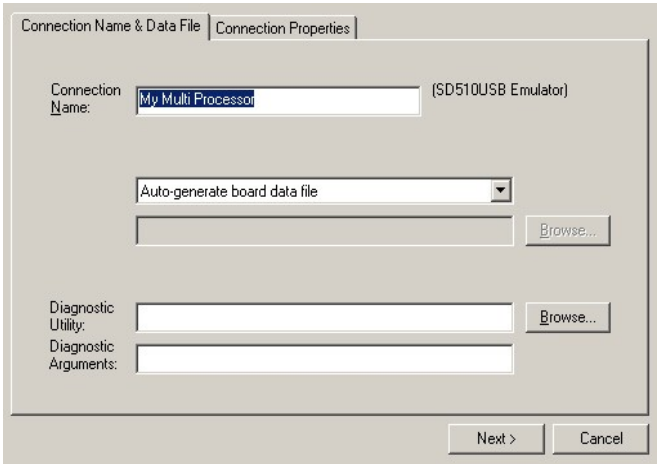
First remove any previous setups in the CCS Setup application.

Add one of the USB SD type driver.



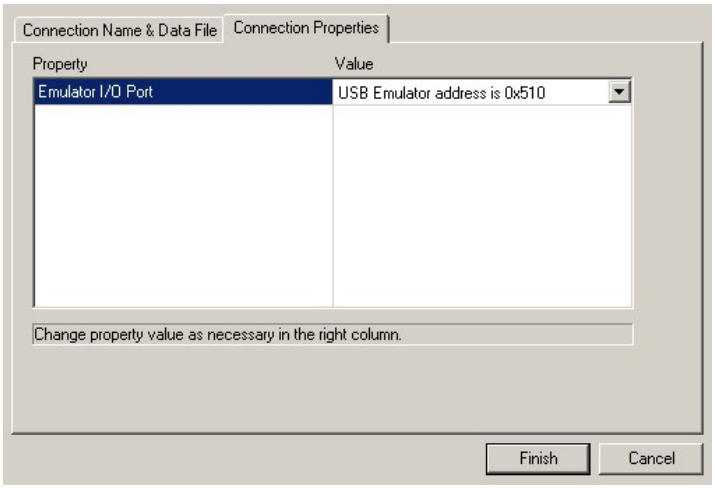
You will see the following screen.

Fill out the name of the board you are using, this can be any name you like.



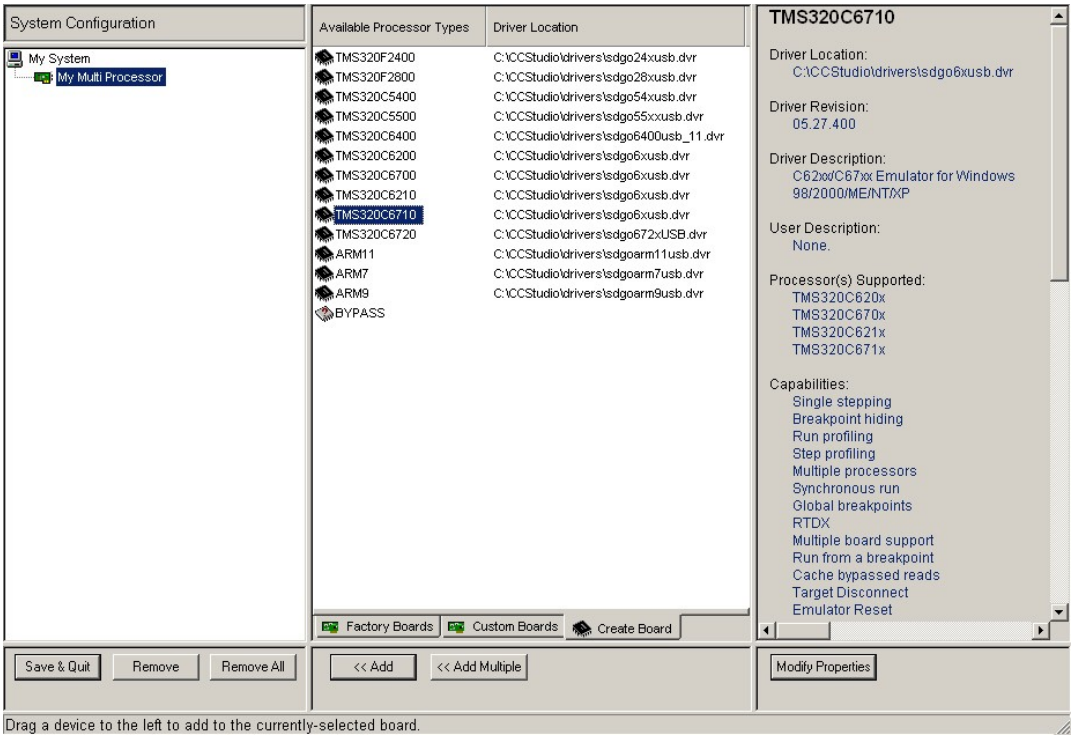
Hit next or move to the next tab

This address should match up with the address in the SdConfig.exe utility

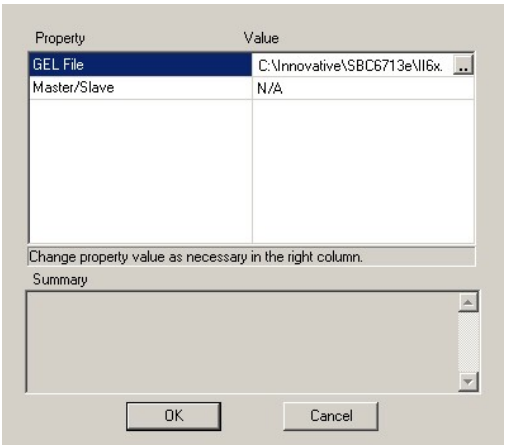


Now we add a processor

Each if the II boards have different processors so match up the closest one for your board.

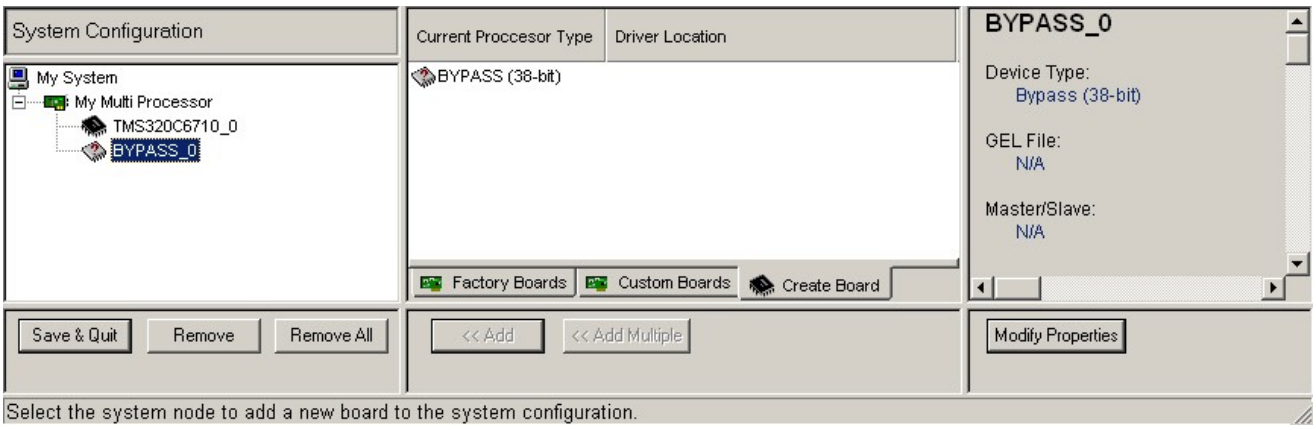


Use the property sheet to find the Gel file from Innovative for your specific board.



Close the processor and choose another processor. This will be a bypass for the DM642. Set the bypass for 38 bits. (For TMS6713 bypass use 42 bits on the first processor, the second processor will be a 64xx and the gel file from II for the DM642). For the Quadia use another C6400 type processor totaling 4 processors. All 4 will use the same GEL file from II.

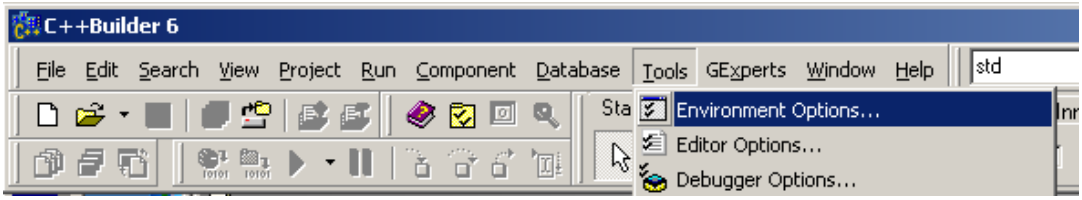
Your system will look similar to this. Save the configuration and quit.



Borland Builder Setup and Use

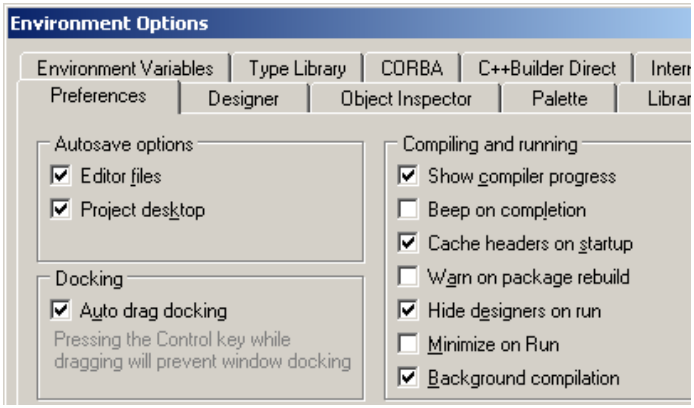
Following the normal installation of the Innovative Integration toolset components, numerous VCL components and C++ classes are automatically added to the BCB IDE. Additionally, Innovative recommends that the following IDE and project options be manually changed in order to insure simplified use and proper operation:

Automatic saving of project files and forms during debugging



31) Select Tools | Environment Options... from the main BCB toolbar.

32) This will invoke the **Environment Options** dialog:



33) Click the Preferences Tab

34) Check "**Editor files**" and "**Project desktop**" under "Autosave Options" so that project files are automatically saved each time a project is rebuilt and debugged.

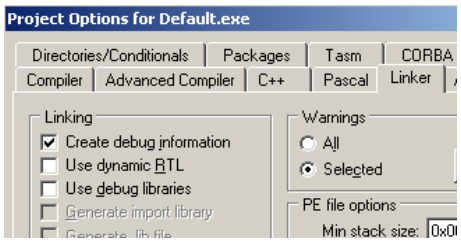
35) Click "OK"

Static-binding of built executables.

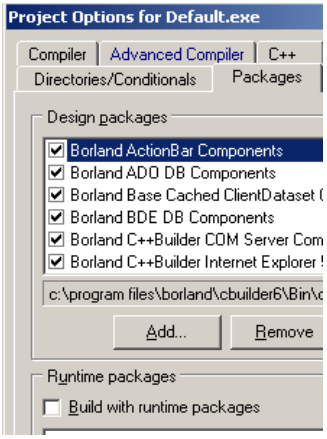
36) Click on Project | Options on the main BCB toolbar to invoke the **Project Options dialog**.

37) Click the **Linker** tab.

38) Uncheck the "Use Dynamic RTL" checkbox.



39) Next, click on the Packages tab and uncheck the "Build with runtime packages" checkbox.



These options insure that projects are built with minimal dependencies on external DLLs. See the FAQ "**What DLLs do I have to deploy with my newly created executable**" in the Troubleshooting chapter for details on which DLLs must be deployed with user-written executables.

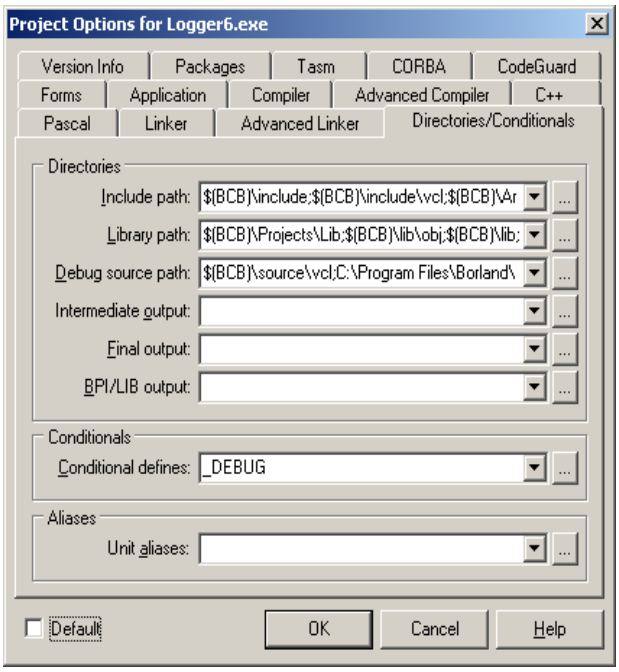
Appropriate library and include paths.

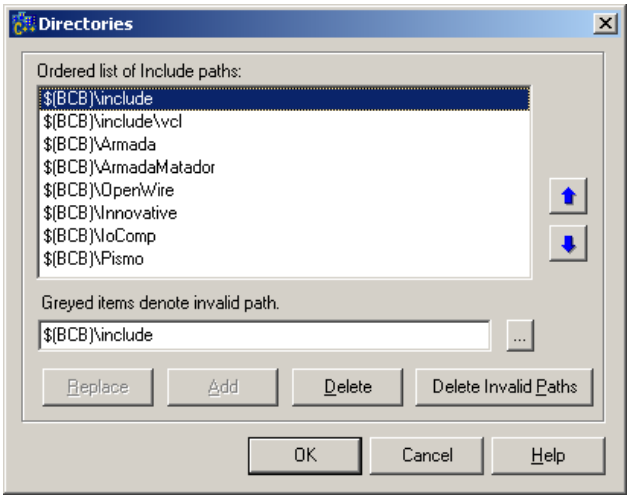
40) Click on the "Directories/Conditionals" tab.

41) Click the ellipses next to the Include Path edit box to invoke the Include Path editor dialog. Add entries for Armada, ArmadaMatador, OpenWire, IoComp and Pismo, as shown below, then click **OK** to accept these edits.

42) Next, click on the ellipses next to the Library Path edit box to invoke the Library Path editor dialog. Add entries for Armada, ArmadaMatador, OpenWire, IoComp and Pismo, as shown below, then click **OK** to accept these edits.

These changes insure that the standard Armada headers and object files are available to projects during compilation. Note that these paths may either be added to the default BCB project, by editing these options without first opening a specific project, or to specific projects after opening them. The advantage of the former is that the settings are automatically present on all subsequently-created projects.


Project Options for Logger6.exe
Version Info
Packages
Tasm
CORBA
CodeGuard
Forms
Application
Compiler
Advanced Compiler
C++
Pascal
Linker
Advanced Linker
Directories/Conditionals
Directories
Include path:
$(BCB)\include;$(BCB)\include\vcl;$(BCB)\Ar
Library path:
$(BCB)\Projects\Lib;$(BCB)\lib\obj;$(BCB)\lib;
Debug source path:
$(BCB)\source\vcl;C:\Program Files\Borland\
Intermediate output:
Final output:
BPI/LIB output:
Conditionals
Conditional defines:
_DEBUG
Aliases
Unit aliases:
Default
OK
Cancel
Help


Directories
Ordered list of Include paths:
$(BCB)\include
$(BCB)\include\vcl
$(BCB)\Armada
$(BCB)\ArmadaMatador
$(BCB)\OpenWire
$(BCB)\Innovative
$(BCB)\IoComp
$(BCB)\Pismo
Greyed items denote invalid path.
$(BCB)\include
Replace
Add
Delete
Delete Invalid Paths
OK
Cancel
Help

SBC6713e Product Migration

Single Board Computer Hardware Features

The hardware complement has changed on production SBC6713e baseboards between revisions B and C. The nature of these changes are summarized below:

Table 1. Production Hardware Changes - Rev B vs C

Peripheral	Rev B	Rev C
C6713 DSP SDRAM size	16 MB	32 MB
C6713 DSP clock speed	225 MHz	300 MHz
DM642 DSP SDRAM size	16 MB	32 MB

As a consequence of these changes, the Pismo support libraries and application programs must incorporate appropriate changes in order to insure proper operation, as detailed below. Once the appropriate changes are made, all libraries and example programs must be recompiled in order to put the changes into effect.

SBC6713e software releases greater than 0.61 have been compiled to accommodate Rev C hardware. Consequently, only users of revision B or earlier hardware need be concerned with implementing the changes below.

Changes to SDRAM initialization.
The larger 32 MB SDRAM devices on the revision C boards utilize a different row format than the 16 MB devices on the earlier board. Consequently, the EMIF Control register value must be changed. Additionally, running the DSP at 300 instead of 225 MHz necessitates a change to the EMIF Timer register.

Table 2. SDRAM Register Changes

Register	Rev B	Rev C
Control	0x57225000	0x53338000
Timer	0x000002ee	0x00000350

These values are referenced within several source files. Each must be edited and recompiled in order to effect the required changes. The affected source files are Innovative\Sbc6713e\HdwLib\IIInit.cpp, Innovative\Sbc6713e\ZenLib\IIInit.cpp and Innovative\II6x.gel. The IIInit() function is affect within the .cpp files, and the Startup() method is affected within the .gel file.

Changes to Phase Locked Loop initialization
The code which initializes the phase-locked loop must be changed to accommodate running at 300 MHz instead of 225 MHz.

Table 3. PLL Register Changes

Register	Rev B	Rev C
PLLM	0x0006	0x0008
PLLDIV3	0x8002	0x8003

These values are referenced within several source files. Each must be edited and recompiled in order to effect the required changes. The affected source files are Innovative\Sbc6713e\HdwLib\IIInit.cpp, and Innovative\Sbc6713e\ZenLib\IIInit.cpp. Only the InitPll() function is affected within these files.

Changes to Application Programs
All DspBios applications include a CDB file to contain static initialization information, including the processor speed, in MHz. Each CDB must be edited using the TI- provided Configuration Editor to modify the CPU speed to 300 MHz when using rev C boards. The value to be edited is located on the System | Global Settings property page.

Upgrading the Firmware.

NOTE: Once you begin the upgrade process, do not turn the power off on the card nor do a hardware reset. Reason: The new firmware will configure new EMIF values and if for some reason you end-up with a pair of Loader / Talker that are not configuring the same EMIF values, the card will not boot.

In this document when making reference to Talker and/or Client, they both refer to the same program.

(1) Changes to Loader program.
The Loader program is in charge of the initial boot up, this program also sets the correct values of EMIF registers for the DM642. It is very important that both Loader and Client programs are in sync (i.e. they both get updated on this release). Below are two ways you can upgrade the client progem.

Option 1. Upgrade running ReLoader.out:

43) Open CCS(r) for CPU_2 (this is the Coprocessor DSP).

44) Do: Debug\Reset CPU. Do: GEL\II Function\Initialize Target\ Execute().

45) Load C:\Innovative\Sbc6713e\Coprocessor\ReLoader.out and run it until it displays “Done.....” in the stdout window of CCS(r).

Option 2. Upgrade Loader using FlashBurn applet.

1) Open FlashBurn.exe applet, type the IP address of your card in the UID box; click on [Connect] button. Since you probably have an old version of the Talker program, the [Connect] button will not turn green, but will stay red.

2) After six seconds, all download buttons will become enabled. Select "Dwld Loader" tab. Now you are in the area that downloads the Loader.bin file. Click on the [...] browse button and go to C:\Innovative\Sbc6713e\Coprocessor\ Loader.bin .

3) Click on [Download Loader] button. Once the "Please wait" message is not displayed, the DM642 coprocessor would have finished downloading the Loader. You may see a message displaying "Bad burn checksum", since your Talker program is not the latest, this message can be ignored.

(2) Changes to the Coprocessor Talker(a.k.a Client)
Several minor bug-fixes and feature additions have been implemented in the coprocessor Talker code. Below are two ways you can upgrade the Talker program (Coprocessor firmware).

Option 1. Upgrade running ReTalker.out:

1) Open CCS(r) for CPU_2 (this is the Coprocessor DSP).

2) Do: Debug\Reset CPU. Do: GEL\II Function\Initialize Target\ Execute().

3) Load C:\Innovative\Sbc6713e\Coprocessor\ReTalker.out and run it until it displays "Done....." in the stdout window of CCS(r).

Option 2. Upgrade using FlashBurn applet.

1) Open FlashBurn.exe applet, type the IP address of your card in the UID box; click on [Connect] button. Since you probably have an old version of the Talker program, the [Connect] button will not turn green, but will stay red.

2) After six seconds, all download buttons will become enabled. Select "Dwld Client" tab. Now you are in the area that downloads the Client.bin file. Click on the [...] browse button and go to C:\Innovative\Sbc6713e\Coprocessor\Client.bin .

3) Click on [Download Client] button. Once the "Please wait" message is not displayed, the DM642 coprocessor would have finished downloading the Client. It takes more to program the Client than the Loader.. You may see a message displaying "Bad burn checksum", since your Talker program is not the latest, this message can be ignored.

*** At this point is a good time to close either FlashBurn.exe applet or CCS(r) and do a hardware reset on the Sbc6713e card. Once you have done this. Continue with step (3).

(3) Changes to Flash Information Section.
The Flash Information Section is a place in flash memory that contains information about the PCB version of the Sbc6713e card plus information regarding the IP (internet protocol) settings of the card. A new field has been added to this section, hence the need of running this program.

1) On rdtx terminal program icon, right click and select properties. Inside the Target box, at the end of the line, please type: -cpu CPU_2 . That line will attach rtdx terminal to CPU_2. We assume that you have name this way in CCS(r) Setup. CPU_1 = C6713, CPU_2 = DM642.

2) Do: Debug\Reset CPU. Do: GEL\II Function\Initialize Target\ Execute().

3) Load and run C:\Innovative\Sbc6713e\Coprocessor\FlashLoad\FlashLoad.out. When the program prompts for revision, please type the revision of the card (i.e. B or C).

Once finished, close CCS(r) and rtds terminal program, and reboot the Sbc6713e card. If you chose DHCP you may find out the IP address of your Sbc6713e card simply bi pinging the Sbc's name. Example: in the DOS prompt type: ping Sbc6713-C101.innovative-dsp.com; this name is provided your card is a Rev C PCB and the serial number is 101. If your PC is not configured for to use DNS, you make have to run an IP scan to find the IP address of your card.

If you chose static IP address, you may ping the card using the IP address you provided.

(4) Changes to the Firmware

Several minor bug-fixes and feature additions have been implemented in the Interface and LAN FPGAs. Use the FlashBurn applet and the supplied Interface and Lan logic images to update the firmware on your baseboard. Below are the instructions.

1) Open FlashBurn.exe applet, type the IP address of your card in the UID box; click on [Connect] button. This time [Connect] button should turn green, unless UID box has the wrong IP address.

2) All download buttons will become enabled. Select "Dwld INTF Logic" tab. Now you are in the area that downloads the interface logic image. There are two types of interface logic depending on the card you have purchased. If the S/N is 115 or greater, then you may have a 600e logic. If you have doubts as to what is loaded on your card, please contact Tech Support. The other kind has a 300e logic device. Click on the [...] browse button and go to C:\Innovative\Sbc6713e\ReLogic600\Sbc6713intf600.exo for a 600e unit or C:\Innovative\Sbc6713e\ReLogic300\Sbc6713intf300.exo for a 300e unit.

3) Click on [Download INTF Logic] button. Once the "Please wait" message is not displayed, the DM642 coprocessor would have finished downloading the image. Here, there is no reason to see "Bad burn checksum", since your Talker program is now the latest.

4) To download the LAN image, switch to "Dwld LAN Logic" tab.

5) Browse to C:\Innovative\Sbc6713e\ReLanLogic\Sbc6713LAN.exo

6) Click on [Download LAN Logic] button. Once the "Please wait" message is not displayed, the DM642 coprocessor would have finished downloading the image. Here, there is no reason to see "Bad burn checksum", since your Talker program is now the latest.

You are done!

There are more features on FlashBurn.exe. Please take a moment to read the section on applets of this manual. Thank you for choosing Innovative Integration products.

About the Baseboard

Single Board Computer Hardware Features

The SBC6713e baseboard features a TMS320C6713 digital signal processor with 32 Mbytes of SDRAM memory.

To complement this core, one or two modular Omnibus I/O modules may installed into the onboard I/O sites. A wide variety of I/O modules are available to address myriad application requirements. The combined baseboard/module system serves a variety of applications including servo applications, data acquisition, stimulus- response measurements and many others.

The tight coupling of the DSP, analog IO and other peripherals make the SBC6713e well-suited for a variety of application such as communications baseband processing, ultrasound applications, multi-axis controllers for high speed servos, RADAR, SONAR applications, communications signal processing and many data acquisition applications.

The baseboard has a variety of features that make it easy to develop high performance systems. On-card, very low noise power supplies provide clean power for analog peripherals installed into either Omnibus I/O site. Other features include 32 bits of programmable digital IO, advanced clocking mechanisms, multi-card triggering and clock sharing plus a flexible timebase for data sampling, and timer/ counters. The SBC6713e features a close to 100 Mb/s Ethernet bus interface for high data bandwidth to the host PC system.

Digital Signal Processor

The SBC6713e baseboard uses the TMS320C6713 DSP, operating at 300 MHz. This DSP is a 32-bit floating point device. The DSP interfaces to the memory and peripherals on the baseboard through its external memory interface (EMIF), which has programmable definitions for the memory interface timing.

DSP External Memory

The SBC6713 baseboard provides 32 Mbytes of SDRAM memory mapped to the '6713 DSP CE0 memory space. This memory provides space for program and data storage as well as storage space for data acquired/generated by the baseboard analog hardware. This memory is programmed to operate at 75 MHz regardless of the DSP core clock rate. The initialization of all external memory spaces are defined within the file `HdwLib\IIInit.cpp` and include the correct parameters for the type of SDRAM used on the baseboard, including refresh timing, as well as timings for all sync and async peripherals and should not be modified.

DSP Initialization

For proper operation of the external peripheral on the baseboard, the external memory interface control registers must be configured prior to use of the external memory interface. Applications built under the Pismo Toolset libraries will automatically initialize the registers appropriately (using code within `HdwLib\IIInit.cpp`). For those customers who need to initialize the registers manually, please refer to the EMIF register initialization values within the `IIInit.cpp` source file to obtain the required register values. If different, the specific values in `IIInit.cpp` supercede those listed in the table below. Please note that the initialization is order sensitive and should be performed in the order given in the table below.

Table 4. 'C6713 DSP EMIF Control Register Initialization Values

Register Name	Address	Value	Use
GBLCTL	0x01800000	0x00003078	
CE0CTL	0x01800008	0x00000030	SDRAM
CE1CTL	0x01800004	0x41A28A22	Asynchronous devices
CE2CTL	0x01800010	0x0000C041	FPDP
CE3CTL	0x01800014	0x21010420	Omnibus Modules
SDTIM	0x0180001C	0x00000350	
SDEXT	0x01800020	0x000544a7	
SDCTL	0x01800018	0x53338000	

During the development process, code may be downloaded to the SBC using a JTAG debugger or via the Ethernet interface. After development is complete, the debugged application image may be placed into Flash ROM so that it will begin execution during SBC power-up.

The DSP reset is controlled by a dedicated reset controller onboard the SBC. When an application program is available in FLASH ROM, the built-in loader software loads the user application software from onboard Flash ROM directly into target memory space as the target powers-up. Software tools are provided to facilitate reprogramming the target application image.

DSP JTAG Debugger Support

Standard TMS320 family JTAG debugger operation is supported by each baseboard. An external debugger connector is supplied that allows use of industry standard JTAG debugger hardware from Innovative, Texas Instruments, and other third party suppliers. The DSP is the only device in the scan path. Software for JTAG debugging and code development is TI Code Composer Studio.

The Pismo Class Library

Innovative Integration's ***Pismo*** is a software class library allows the developer to fully exploit the advanced hardware features of the Innovative DSP product lines and to reap all the benefits from Texas Instrument's DSP/BIOS Operating system. Every board peripheral has been carefully integrated into the OS and its functionality encapsulated in a device driver

that can readily be controlled within DSP/ BIOS applications including PCI interface, analog I/O, external bus and memory, serial ports and other I/O devices.

Pismo provides extensive C++ class support for:

- Dynamic creation and runtime control of tasks
- Simplified management of and access to all TI Chip Support Library (CSL) and DSP/BIOS API functions including: Semaphores, Mutexes, Mailboxes, Timers, Edma, Qdma, Atoms, McBsp, Timebases, Counters, etc.
- Foundation (base) classes for DMA-driven device driver development
- Templatized queues
- Partial standard-template library functionality via STLPort

For example, the code fragment below uses the Pismo IntBuffer class to initialize a QDMA (quick DMA) to perform a memory-to-memory move of a constant value (0) into a 4096-word buffer (at Src), then to copy the source buffer (Src) to the destination buffer (Dst):

```
// Create a source buffer of 0x1000 integers
IIBuffer Src(0x1000);
// Initialize the source buffer with zeros
Src.Set(0);
// Create a destination buffer of 0x1000 integers
IIBuffer Dst(0x1000);
Dst.Copy(Src);
```

Simple To Use

In the same way, peripheral-specific class libraries dramatically simplify access to board-specific peripheral features. For example, the code fragment below illustrates real-time processing and display of analog input signals running on the SBC6713e DSP board equipped with an Omnibus module within a separate thread of execution:

```
//--------------------------------------------------------------------------
//  LoopThread() --  Capture snapshots of A/D input
//--------------------------------------------------------------------------

class LoopThread : public Thread
{
public:
    LoopThread(IIPriority priority)
        : Thread(priority), FCount(0), Cursor(0), Requested(false)  { }

    int             Count()
        {  return FCount;  }
    void            Resize(int size)
        {  CaptureEvents=size;  Snaps.Resize(size);  }
    IntBuffer &     Acquire()
        {
        Cursor = 0;
        Requested = true;
        Available.Acquire();

        return Snaps;
        }
```

```
protected:
    // Fields
    volatile int    FCount;

    // Data
    bool            Requested;
    Semaphore       Available;
    IntBuffer       Snaps;
    int             Cursor;
    int             CaptureEvents;

    // Methods
    void Execute()
        {
        // echo input to output
        while(!Terminated())
            {
            AIn.Get();
            ++FCount;

            //
            // If main thread wants a block, copy it to him
            if (!Requested)
                continue;

            int Residual = Snaps.Ints()-Cursor;
            int Chunk = std::min(Residual, AIn.Buffer().Ints());
            Snaps.Copy(AIn.Buffer(), Cursor, Chunk);
            Cursor += Chunk;

            if (Cursor == CaptureEvents)
                {
                Requested = false;
                Available.Release();
                }
            }
        }
};
LoopThread Loop(tpHigher);
```

Not Just for C++ Experts
Note that even if you're not a C++ maven, the code is quite clear and understandable. In fact, one of the benefits of using C++ is that while it helps to mitigate and manage complexity to support creation of larger, more sophisticated applications, it is often simply used as a "better" dialect of the C language. C++ is essentially a superset of C. As such, you may freely intermix calls to legacy 'C' functions, newly-written C functions, Assembler functions and C++ functions (called methods) within C++ programs. You need not fully understand all of the enhanced capabilities and features of C++ in order to fully exploit the features of the class libraries provided in Pismo.

Analog I/O Streams

The Analog I/O is, for most applications, the most important feature of the SBC6713e baseboard. Most of the peripherals on the hardware are related to Analog I/O. Most of the configuration options are related to Analog I/O. It is the part that causes the most problems in development.

To maximize the chances for success, the Pismo library provides a set of classes that hide all of the details of data acquisition. From the application level, the user simply processes buffers of data. The details of hardware and software management are isolated from the application.

Stream Objects and Device Drivers

Data I/O in DSP/BIOS is accessed and controlled via custom device drivers. Access to the device driver is controlled by a Stream class. These drivers are dynamically installed by the Stream when needed by the user application. From the point of view of the application the stream control class provides all of the user interface function needed to configure and operate the I/O operation.

Table 5. Device Driver and Stream Classes

Device Driver Class	Stream Class	Description
AnalogInputDriver	AnalogInStream	Streamed Input from an analog source. Continuous data flow with buffering. Data flow stops only via trigger control.
AnalogOutputDriver	AnalogOutStream	Streamed Output to an analog source. Continuous data flow with buffering. Data flow stops only via trigger control.
CaptureInputDriver	CaptureInStream	Burst Input from an analog source. Data flow is discontinuous, filling each buffer requested and stopping.
ServoBase	ServoIntf	Continuous, low-latency analog capture and playback suitable for performance servo control applications. Event-at-a-time application data processing.
Stream	FpdpInStream	Burst Input from the FPDP hardware.
Stream	FpdpOutStream	Burst Output to the FPDP hardware.

Hardware Isolation and Independence

The Analog and the Capture driver allow a single Stream to be used with different analog hardware. In the SBC6713e, for example, Omnibus modules allow a wide variety of analog choices on a single baseboard. Each of the Stream classes can be “attached” to a particular module and will automatically configure itself to use that hardware.

The Servo driver provides low-latency, interrupt-driven data processing, suitable for real-time control applications, albeit at the expense of high CPU usage. It is currently implemented only for the Servo16 module.

The FpdpInStream and FpdpOutStream drivers provide communications with external Front Panel Data Port devices.

Stream I/O Types

There are two distinct categories of Streams implemented within Pismo - Continuous and Burst.

Continuous Streams use the model that the input or output is a continual process, whether periodic or not. Thus in order to avoid data loss when the application is momentarily busy, internal buffering is provided so that the hardware may operate for extended periods without software intervention. This means latency must be increased: data may be ‘in the queue’ for some time until additional data forces it out to the application. Of course, if the application does not process the data as fast as it arrives data will eventually be lost.

Burst Streams use a different model. Here data movement is 'on demand' instead of asynchronous. If no request for action by the application is received, the Stream is idle. This type of Stream is more common for non-Analog I/O such as the FifoPort or PCI busmastering, but the CaptureInStream implements a burst type I/O model on the analog hardware.

Stream Buffer Model
Each Stream uses data buffer class objects to pass data between its hardware and the application. These buffers are all the same size. Passing data between Streams is simple if the buffers are chosen to be the same size:

```
Ain.Get();
Aout.Put(Ain.Buffer());
```

Buffer transfers are efficient because the data buffers are not copied at any time during the transfer process. By default, Streams allocates three buffers, two internal and one swap buffer. If desired, the number of internal buffers in the pool may be modified prior to opening the Stream by assigning a new value using the BufferCount method.

```
// Instantiate the analog stream objects
AnalogOutStream Aout;
Aout.BufferCount(5);
AnalogInStream Ain;
Ain.BufferCount(5);
```

This code forces the Stream to allocate five internal buffers.

The size of data buffers may be specified explicitly using the Stream::Events method. This latter method sizes the buffers such that they can contain the at least the specified number of acquisition "events", where an event is defined as one sample from all enabled A/D or D/A channels. This simplifies most buffer processing algorithms since all buffers are guaranteed to contain an integral number of samples from all enabled channels.

The product of the buffer size and the number of buffers gives the load-carrying capacity of the system. For example, the originally allocated three buffers per stream, each sized at 0x1000 bytes, running at 44.1 kHz equates to a load carrying capacity of

```
(0x1000 bytes/buffer) x (3 buffers) / (44100 samples/sec) /
(2 bytes/sample) = 139 mS
```

Whereas in the second example, with six buffers per driver pool

```
(0x1000 bytes/buffer) x (6 buffers) / (44100 samples/sec) /
(2 bytes/sample) = 278 mS
```

Data integrity can thus be preserved at the expense of additional memory utilization.

Burst Streams place data into the buffer provided. No buffering is used, and data acquisition is halted when the provided data buffer is filled.

Stream Internals
The DSP CPU used on the SBC6713 is powerful and fast, yet the Stream classes improve performance even further by drastically reducing CPU use for data movement. The available DMA channels in the C6000 DSPs are fully exploited to do movement to and from hardware to memory so that hardware interrupt rates rarely exceed 1KHz! The net effect is that virtually all of the bandwidth of the CPU is available for application processing, without requiring any application DMA programming.

Multitasking Friendly
The Stream classes support efficient cooperation in multitasking applications. Any function that requires a delay to complete. will block using DSP/BIOS functions that release other OS threads for efficient utilization of the processor.

Using Analog Streams in an Application

The AnalogInStream, AnalogOutStream, and CaptureInStream all allow fast data movement between the application and the hardware in different modes. Once associated with a hardware device, they allow all configuration and control of the session to take place through the methods of the Stream.

Every Stream must consider these questions to make a functioning application:

- Which stream to use (Input vs. Output or Continuous vs. Burst).
- Which hardware to use, and in which hardware mode.
- Which clock source to use, and with what parameters.
- Which triggering mode to use, and with what parameters.

Once these are taken care of, using the Stream to perform the Analog I/O is a simple matter. Consider the code fragment below which illustrates all of the steps necessary to fully initialize and stream a stream a continuous 1 kHz sine wave to the analog outputs present on a SD16 Omnibus module attached to an SBC6713e baseboard SD16 module at 50 kHz:

```
using namespace II;

//-------------------------------------------------------------------------
//  IIMain() --  Illustrate Omnibus Output
//-------------------------------------------------------------------------

AnalogOutStream  Aout(Omnibus::mSite0);

void IIMain()
{
        // ...Load Module onto site 0.
LoadModule(Omnibus::mSite0, Omnibus::mtSD16);
        //
//  ...Use default clock (DDS)
        //  ...Set Clock Rate
ClockRateUIPtr(Aout.Clock())->Rate(50000);
        //
        //  ...Output on all channels
Aout.Channels()->EnableAll();
        //...Size the buffers
        Aout.Events(5000);
```

```
Aout.BufferCount(BuffersPerSec/3);

        // Open the streams
        Aout.Open();

        // Stream loop...
        bool run = true;
        while ( run )
                {
                Aout.Generate();
                Aout.Put();
                }

                // Terminate streaming
        Aout.Close();
}
```

Examining the above code, you can see the application uses an AnalogOutStream, since it is an output program. The next interesting line is the call to LoadModule(). This informs the system of which module is 'plugged in' on Omnibus::mSite0, where the stream is also attached. With the attachment of the module to the stream, the stream object can configure hardware, clock, and trigger settings.

The next several lines configure the Stream clock rate, the channel configuration, and the stream buffer count and buffer size. Then comes the Stream::Open method which activates the device driver in DSP/ BIOS. Afterwards, the Stream::Control method may be used to perform any necessary device-specific initialization and/or control functions.

Data flow begins with the calls to Generate() and Put(). Generate() uses signal generator classes to write a signal pattern into the buffer. Put() enqueues the data for output. Data output will not actually begin until the buffer queue is essentially filled with data. This avoids under-runs of the output. For input Streams, the Get() method starts streaming at the first call. These buffer methods should be repeated to keep the streams flowing. After use of a device is complete, it is closed using the Stream::Close() method.

Note that in the above example, the type of module used matters very little in the finished application. Simply by changing a single constant this code can be rebuilt to work on any module that supports Analog output. The module specific details are handled by the Pismo library internally.

Selecting the Stream Object

Each Stream object is used to manage input or output on a single Omnibus module. Multiple module applications need to use separate instances of the stream for each module site. The Stream object is associated with a module site by the constructor, allowing access to the hardware for configuration:

```
    AnalogInStream  AIn(Omnibus::mSite0);
```

The `AnalogInStream` provides continuous streaming input. All data is delivered to the ring of internal buffers and from there to the application.

`AnalogOutStream` allows continuous streaming output to an output device. The application must deliver data as fast as it is consumed to avoid buffer underruns.

The `CaptureInStream` is a new type of driver that is used to emulate manual capturing of data to a buffer. In this mode, the analog hardware is inactive until a buffer is presented for filling. When this occurs, an acquisition is started that will fill the presented buffer, after which data taking is stopped. The process repeats for each buffer presented.

In the capture mode, no processing is taking place when data is not requested. This is a major difference from `AnalogInStream`. Also, if the buffer is sized to be smaller than the analog hardware's own FIFO or storage, each buffer will be a snapshot dump of the FIFO contents. This makes capture useful for snapshots of very high rate analog input, faster than the module can be read. There is no way to take continuous data sets larger than a single buffer in capture mode. There will be a gap between any two captures. `AnalogInStream` should be used for continuous applications.

Selecting and Configuring Hardware

The Sbc6713e supports Omnibus Modules, allowing multiple hardware configurations on the same baseboard. Modules are attached to an Omnibus site by a call to LoadModule().

```
// ...Load Module onto site 0.
LoadModule(Omnibus::mSite0, Omnibus::mtSD16);
```

Once a module is loaded, it can be accessed by the stream object's Module() method. This returns a generic pointer that can be converted to an exact module pointer to allow its methods to be called. The functions that do this are called Module Conversion Functions.

Module Conversion Functions are provided for all supported Omnibus modules. See the online help for the module class for a description of these functions. An example of the use of these functions:

```
SD16 * sd16 = SD16Ptr(AIn.Module());  // Gives valid SD16 object
A4D4 * a4d4 = A4D4Ptr(AIn.Module());  // Returns 0 -- not an A4D4!
```

Note that a module conversion function will fail if the conversion can not take place.

Selecting and Configuring Clocks

On attachment of the stream to a module, the Clock configuration system becomes active. It consists of the following methods:

Table 6. Stream object Clock Methods

Method	Description
IsClockSourceSupported()	Returns True if a clock source is allowed on the hardware.
SetClockSource()	Change clock source to the selected source.
Clock()	Returns an interface object that allows the configuration of the Clock source.

The User Interface object returned by the Clock() method is used to configure the selected clock source. The possible ways a clock can be configured depends strongly on the type of source. For example, an internal clock such as the DDS can have the

clock frequency programmed. An external clock can not. Rather than provide a complicated set of functions, many of which may not work for a clock source, we instead separate each distinct part of the User Interface into separate interface classes. The interface object for a clock source may support none, or one, or any number of all the possible UI interface classes. An interface can be accessed by the conversion function for each of the UI interface classes. If an interface is not supported, the conversion function returns a null pointer.

The following code sample shows the use of conversion functions and multiple interface classes. The DDS clock source is in use. It supports both the ClockRateUI interface, which allows changing the clock frequency, and the ClockSyncUI interface, which allows configuration of the SyncLink/ClockLink master hardware to drive the DDS clock signal off the baseboard for use as a source on another board.

```
//  ...Set Clock Rate (allowed on DDS)
ClockRateUIPtr(AIn.Clock())->Rate(50000);
ClcokSyncUIPtr(AIn.Clock())->SyncLinkChannel(scSyncLink0);
```

Each line can be read from the inside out. AIn.Clock() returns the current clock UI object. This object is input into the clock conversion function ClockRateUIPtr() and converted into the ClockRateUI interface. Finally, the Rate() method of this class is called to set the rate of the clock to 50,000 Hz.

Selecting and Configuring Triggers

The Analog Stream objects allow the user to configure the triggering method used during the run. Triggering features are divided into four parts:

- Pretriggering - Handling data before the start trigger.
- Start Trigger - How to start data taking.
- Stop Trigger - How to stop data taking.
- Retriggering - How to handle interval before next start trigger

Selecting Pretriggering Modes

The pretrigger control for a stream consists of the following methods:

Table 7. Stream object Pretrigger Methods

Method	Description
IsPretriggerTypeSupported()	Returns True if the pretrigger mode is allowed on the hardware.
SetPretriggerType()	Change pretrigger to the selected mode.
Pretrigger()	Returns the pretrigger interface for the mode.

Table 8. Stream object Start Trigger Methods

Method	Description
IsStartTriggerTypeSupported()	Returns True if the start trigger mode is allowed on the hardware.
SetStartTriggerType()	Change start trigger to the selected mode.

StartTrigger()	Returns the start trigger interface for the mode.

Table 9. Stream object Stop Trigger Methods

Method	Description
IsStopTriggerTypeSupported()	Returns True if the stop trigger mode is allowed on the hardware.
SetStopTriggerType()	Change stop trigger to the selected mode.
StopTrigger()	Returns the stop trigger interface for the mode.

Table 10. Stream object Retrigger Methods

Method	Description
IsRetriggerTypeSupported()	Returns True if the retrigger mode is allowed on the hardware.
SetRetriggerType()	Change retrigger to the selected mode.
Retrigger()	Returns the retrigger interface for the mode.

Trigger configuration presents the same problem as the clock configuration, except more so. Triggering consists of four parts, each of which can be independently set. In addition, there are far more ways of defining triggers than there are for defining parts of a timer. For example, the start of data flow on Omnibus modules can be based off of an external digital signal, the value of the data on an input channel, by software command, or be automatic. Pretrigger and Stop trigger options are also numerous. A class that had methods for all these features would be large, complex, and would usually have most of its functions inoperative without giving the application any feedback.

Trigger configuration is also complicated by Module differences. Even the same type of trigger can differ in on different modules. For example, an external start trigger may allow the condition of the input signal (edge or level triggering, positive or negative polarity) be changed. Another module might not support changing these features, being always positive edge triggered. Another example is that threshold triggering might be able to be triggered of any selected channel, or it might be restricted to a predetermined channel.

The triggering configuration system uses trigger interface objects to allow configuration of the triggering modes. There are four access functions (Pretrigger(), StartTrigger(), StopTrigger(), and Retrigger()) giving interface objects for use. Each of these objects supports none, or one, or any number of Trigger UI interface classes for configuration, each of which can be exposed by a conversion function for the class. If a UI interface is not supported, the conversion function returns a null pointer.

The following code sample shows the use of trigger conversion functions and multiple interface classes. The AD40 module is in use. It supports several different Pretrigger and StartTrigger modes. In the example we wish to use Counted Pretriggering and Threshold Start Triggering.

```
//
//  This example for an AD40 uses pretriggering and
//     start triggering...
//
//  Set triggers
Stream->SetPretriggerType(TriggerManager::ptCounted);
Stream->SetStartTriggerType(TriggerManager::stThreshold);
//
//  Configure Pretrigger to 500 counts
```

```
CountedPretriggerPtr(Stream->Pretrigger())->PretriggerCounts(500);
//
//  Set start threshold to .1 volts
VoltageThresholdPtr(Stream->StartTrigger())->ThresholdLevel(.5);
ConfigurableTriggerPtr(Stream->StartTrigger())->Type(ttEdge);
ConfigurableTriggerPtr(Stream->StartTrigger())->Polarity(tpPositive);
```

Counted Pretriggering allows the preservation of data samples before the start trigger fires. Normally the first data point read was taken just after the start trigger fires. With Counted Pretriggering, N samples before the trigger fires are output when the trigger fires. In the example below the pretrigger is set to return 500 samples from before the trigger.

The AD40 Threshold mode supports two interfaces: VoltageThreshold and ConfigurableTrigger. VoltageThreshold configuration allows the threshold to be set in volts with the ThresholdLevel() method. In the above example the hardware is configured to trigger at half a volt. The ConfigurableTrigger UI interface allows signals to be specified as Type edge or level, and Polarity positive or negative. In the example above, the trigger is configured for a positive-going edge. This means that when the signal crosses the threshold from below to above, the start trigger will fire. A crossing from above the threshold to below it will not fire the trigger.

Each line can be read from the inside out. Stream->StartTrigger() returns the current start trigger UI object. This object is input into the trigger conversion function ConfigurableTriggerPtr() and converted into the ConfigurableTrigger interface. Finally, the Type() method of this class is called to set the trigger type to ttEdge.

The trigger modes a module supports and the UI interfaces its supported modules support are very module dependent. It is quite common to have to use several conversion functions to configure a trigger mode. It is also common for a trigger to be unconfigurable, exposing no trigger UI classes. Similarly, many modules support several triggering modes. Other modules support only the default, unconfigurable combination of no Pretriggering, Always start, Never stop, and no Retriggering. The description of the modes a module supports and the UI interfaces a module supports in each mode are listed in the online help with the description of each module.

Interrupt Handling

In DSP/BIOS, all hardware interrupts are intended to be managed by a DSP/BIOS hardware manager. This manager allows user functions to be called as part of the interrupt process while still cooperating with DSP/BIOS. As a part of the configuration process, the user can direct the HWI manager to call a user function.

Interrupts in a C++ Environment
In a system using C++, this means of attaching interrupts leads to several difficulties. A minor problem is that of name-mangling. C++ creates a new name for every function created in order to allow overloaded functions. The DSP/BIOS configuration does not understand the new name and results in a linker error. There is a simple work-around for this:

```
extern "C"
{
    void  MyHandlerFunction( void * arg );
}
```

This declares to the compiler to create a standard C symbol name for this function (`_MyHandlerFunction`) which can be used by to the DSP/BIOS configuration tool.

A more fundamental problem is that this mechanism does not allow the interrupt handling function to be changed during the life of the program. Also, this handler function may not be a class member function. This restriction can make designing a class object that handles interrupts awkward.

The Pismo Solution

The solution implemented in the Pismo environment is to take over all interrupt handling by providing a full set of standard handlers. The user then never needs to work in the CDB editor to provide handlers. The standard Pismo handlers contain code that will call a user's installed interrupt handler function if one is provided. While this adds a small amount of latency to the interrupt, the DSP/BIOS overhead per interrupt call is still much greater and dominates the total time per interrupt. In general, the BIOS environment is not suited for extremely high interrupt rates. Luckily, the use of DMA to acquire data from FIFOs on peripherals means that high rate interrupt handlers are not needed.

Pismo uses a special object, a Binder, to group a handler function and its arguments in a way that can be properly called by the standard handler. One form of Binder is used to attach a stand-alone function and its arguments, another form allows the binding of an Object, a member function of that object, and its arguments. This form of binder can allow a class object instance variable to act as a handler for interrupts. Here is an example from the Messages example of defining a binder for a timer interrupt:

```
//
//  Timer Interrupt Handler Function
void  OnTimerFired(int arg);
//
//  Binder Object for Timer
typedef void (*IntFtnType)( int arg );

FunctionHandler<IntFtnType, int> TimerBinder(OnTimerFired, 0);
```

And attaching the binder to an interrupt:

```
//  Set up a real time clock to send commands to host on
//    Target channel...
Irq Timer0( intTimer0 );
Timer0.Install( TimerBinder );
Timer0.Enable( false );
//
//  Turn on the clock at 5 hz
DspClock  Tclk0(50.0, 150.0);

Timer0.Enable( true );
```

In the example, TimerBinder is an object that collects the handler function, OnTimerFired, and its argument, 0. This object is passed into an Irq object associated with the TCLK0 interrupt. When the timer interrupt fires, the handler will be called with its argument. The binder is a template, allowing any type of argument to be used with an interrupt handler.

Class Irq

Class Irq is an object that can be created to manage a specific interrupt. It has functions to set, clear, enable and disable the interrupt and also allows a handler to be installed that will be called whenever the interrupt fires. In the above code, see how all functions involving the interrupt were encapsulated in the methods of the Timer0 class object.

Interrupt Lock Classes

A common need in a program is the ability to disable a particular interrupt, or all interrupts, in a portion of the program. The standard means of standalone functions (an disable followed by a enable interrupts) has a few problems. The first is that the means does not nest well. If a function blocking interrupts is nested in a second one, interrupts will be re-enabled at the wrong time. A second is that if the function has multiple return paths, each must have the re-enable code in it. The introduction of C++ exceptions makes this problem even worse.

The Pismo library provides a set of class objects that meet this problem. These lock objects disable a particular interrupt or all interrupts in a region and restore the state to what it was on entry when the lock object is destroyed. If the object is created on the stack, any means of exiting the block in which the object is defined will cause the cleanup code to be called. Calls to these objects properly nest as well.

Table 11. Interrupt Lock Classes

Lock Class	Interrupts Affected	TI Class Library
InterruptLock	One IRQ	CSL.
GlobalIntLock	All interrupts	CSL.
HwiGlobalIntLock	All interrupts	DSP/BIOS.

Interrupt Binder Templates

The Binder system can be thought of as a more flexible and powerful version of a function pointer variable, allowing a user callback function to be called indirectly without knowing more than the interface to the function. Since the binder objects are templates, the type of the function and its arguments are not fixed but can be of any type. Also, member functions can be bound to an interrupt, which a callback function can never do.

The Binder system is powerful, yet in practice is quite simple to use. This system illustrates the power of the C++ language to contain a complicated system in a simple-to-use package.

Class InterruptHandler

This class is a base class for the ClassMemberHandler and FunctionHandler templates. It provides the interface the Pismo system uses to call the interrupt handler.

Class ClassMemberHandler Template

This template allows the binding of a member function of a class object with the object to call and an argument of any type. In this example the IsrHandler class is bound to a timer interrupt:

```
class IsrHandler
{
```

```
public:
    IsrHandler()
        : Binder(*this, &IsrHandler::MyHandler, &Tally), Tally(0)  { }
    ClassMemberHandler<IsrHandler, unsigned int *> Binder;

    void MyHandler(unsigned int * tally)
    {
        *tally += 1;

        if ((*tally & 0x7f) == 0)
            rtdx << "Isr tally: " << *tally << endl;
    }


private:
    // Data
    unsigned int    Tally;
};

// Instantiate a concrete instance of above class..
IsrHandler  Isr;

void IIMain()
{
    // Dynamically create an Irq object tripped from onchip timer 0
    Irq Timer0( intTimer0 );
    // Bind and install the interrupt vector
    Timer0.Install( Isr.Binder );
    // Program onchip timer 0 to signal at 100 Hz
    Timer0.Enable( false );
    DspClock Clock(100, 150, true, 0);
    Timer0.Enable( true );

    // Use RTDX event log to monitor progress
    rtdx.Enabled(true);

    rtdx << "Message from within IIMain,,,"<< endl;

    // Go to sleep...
    while (1)
        TSK_yield();
}
```

In the above example, the handler uses a int * argument to pass out information from the interrupt routine.

Class FunctionHandler Template

This template allows the binding of stand-alone function with an argument of any type. In this example the OnTimerFired function is bound to a timer interrupt:

```
//
//  Timer Interrupt Handler Function
void  OnTimerFired(int arg);
//
//  Binder Object for Timer
typedef void (*IntFtnType)( int arg );

FunctionHandler<IntFtnType, int> TimerBinder(OnTimerFired, 0);
```

This is the installation of the handler in the program:

```
//  Set up a real time clock to send commands to host on
//    Target channel...
Irq Timer0( intTimer0 );
Timer0.Install( TimerBinder );
Timer0.Enable( false );
//
//  Turn on the clock at 5 hz
DspClock  Tclk0(50.0, 150.0);

Timer0.Enable( true );
```

EDMA and QDMA Handling

The TI C6000 processor supports a rich, powerful DMA engine to move data without CPU intervention. There are two kinds of DMA allowed. One, EDMA is full featured but can take some time to set up. QDMA is TI's facility for quick DMA movement of data. It is similar to a normal DMA transfer except that it is software triggered and performs only a single transfer. No linking of blocks is permitted with QDMA. It also is faster to initiate as only a few registers need to be set to start a new transfer.

Both kinds of DMA use a set of registers to define the configuration of a DMA transfer. By properly configuring the settings, many different transfer types can be performed, such as interelaved data, two dimensional arrays, and so on. See the TI Peripheral Library guide for more information on configuring EDMA and QDMA.

The QDMA has a single set of configuration registers, so only one QDMA may be in progress at the same time. The EDMA has a pool of blocks that may be used to define simultaneous, complex transfers.

Class DmaSettings
The DmaSettings class manages an image of the settings registers used to configure a QDMA or EDMA transfer. It provides properties to read and set the individual fields of the registers, saving the user the effort of masking bits and shifting data. It even provides functions that preconfigure some commonly used transfers, saving even more programmer effort.

The following code fragment shows how the setter functions are used to set up for a transfer. The DmaSettings class returns a reference to self on all setter functions, allowing multiple parameters to be set on a single line:

```
DmaSettings Cfg;

Cfg.Priority(DmaSettings::priHigh).ElementSize(DmaSettings::is32bit)
Cfg.SourceIncr(DmaSettings::Incr).DestinationIncr(DmaSettings::Incr);
Cfg.TCInt(true).TCCode(1).FrameSync(true);
Cfg.SourceAddr((int)&src_array[0]).DestinationAddr((int)(dest_array+50));
Cfg.ElementCount(50).ElementIndex(1);
Cfg.FrameCount(0).FrameIndex(1);
```

Class Qdma
This class manages the posting of Qdma requests. It contains functions to allow configuration of a transfer, initiating a transfer and completion notification via either an interrupt or a polling function. Because the system state is saved in the object, transfers can be predefined and saved to be posted at a later time.

As with all DMA objects, the Qdma object uses an internal DmaSettings object to define the transfer. The Settings() method provides access to the object to allow calling the DmaSettings classes own configuration functions, or configurations can be loaded from a second object with the Load() method.

```
// Q is a Qdma object, here we change the destination address
Q.Settings().DestinationAddr((int)(dest_array+0x10));
```

For QDMA, a transfer is initiated when the parameters are loaded into the QDMA registers. This is performed by the Submit() method, which starts the preconfigured transaction, or loads the passed in configuration and submits it.

Only one Qdma transfer may be active in the system at one time. Multi-threaded applications must arbitrate Qdmas as appropriate.

If a terminal count interrupt is not used, a call for WaitForComplete() will delay until the completion occurs. TestComplete() will return a flag that can be used to check completion without blocking.

Qdma transfers may be configured to generate Terminal Count interrupts on completion of the transfer. Which TC bit is signaled is configured in the settings block.

A user supplied handler, similar to an interrupt handler, can be associated with the terminal count interrupt by a call to the TcIntInstall() method. The DMA system shares a single interrupt for all TC interrupts, and the system will call the installed handler when the particular bit in the TC register becomes set. The handler installer requires an Interrupt Binder Object (See " Interrupt Binder Templates " on page 54.) as an argument to associate a handler function or method and argument for the interrupt forwarding mechanism of Pismo. A second function, TcIntDeinstall() removes any installed handler.

Once installed, TC interrupts may be enabled or disabled by a call to TcIntEnable().

The following example shows a full Qdma transfer with TC interrupt handling. In this example a class member function is bound to handle the interrupt response.

```
class DmaIsr
{
public:
    typedef void (*IntFtnType)(void * fallow);

    DmaIsr()
        : Binder(*this, &DmaIsr::MyHandler, NULL)  {  }

    void MyHandler(void * fallow)
    {
        qdma_not_done = false;
    }

    ClassMemberHandler<DmaIsr, void *> Binder;
};

DmaIsr Isr;

void IIMain()
{

    DmaSettings Cfg;
    Cfg.Priority(1).ElementSize(0).SourceIncr(1).DestinationIncr(1);
    Cfg.TCInt(true).TCCode(0);
```

```
        Cfg.SourceAddr((int)src_array).DestinationAddr((int)dest_array);
        Cfg.ElementCount(100).ElementIndex(1);
        Cfg.FrameCount(0).FrameIndex(1);
        Qdma Q(Cfg);

        // This QDMA operation will trip a terminal count interrupt when
        // all data has been moved.
        Q.TcIntInstall( Isr.Binder );
        InitArrays();
        Q.TcIntEnable(true);
        qdma_not_done = true;
        Q.Submit();
        while (qdma_not_done)
            ;
    }
```

Class Edma

This class manages the posting of EDMA requests. It contains functions to allow configuration of a transfer, initiating a transfer and completion notification via either an interrupt or a polling function. Because the system state is saved in the object, transfers can be predefined and saved to be posted at a later time.

An additional feature of EDMA is the ability to build complicated transfers by linking EDMA transfer blocks or by chaining EDMA transfers together.

For more information on EDMA, see the TI Peripheral Guide.

As with all DMA objects, the Edma object uses one or more internal DmaSettings object to define the transfer. One block is allocated for the primary transfer, and one for each linked block. The Settings() method provides access to the primary transfer block's settings object. The LinkSettings() similarly allows to one of the link blocks's DmaSettings object. Each of these can be used to call DmaSetting's own configuration functions, or configurations can be loaded from a second object with the Load() method.

```
        // Ed is a Edma object, here we change the destination address
        Ed.Settings().DestinationAddr((int)(dest_array+0x10));
```

The EDMA transfer can be attached to one of a number of channels. To attach an EDMA to a hardware interrupt, use the channel with the same number as the hardware interrupt. For example, to attach an EDMA to external interrupt 4, use the EDMA channel 4.

For EDMA, before a transfer can be initiated, the parameters are loaded into the EDMA PRAM registers. This is performed by the Submit() method, which loads the PRAM with the transfer information. Unlike QDMA, this does not start the transfer itself. The transfer will be initiated when the associated hardware interrupt occurs. If using software triggering, use the Set() function to initiate a transfer. One Set() call is required for each link block in the transfer.

Each Edma transfer allocates blocks from the PRAM pool to configure its Link blocks. These blocks are a limited resource, and the allocation may fail. If the failure occurs, the IsValid() function will return false.

If a terminal count interrupt is not used, a call for WaitForComplete() will delay until the completion occurs. TestComplete() will return a flag that can be used to check completion without blocking.

Edma transfers may be configured to generate Terminal Count interrupts on completion of any and all blocks in the transfer. Which TC bit is signaled is configured in each settings block. This means there can be different handlers for different blocks in the transfer.

A user supplied handler, similar to an interrupt handler, can be associated with the terminal count interrupt by a call to the TcIntInstall() or LinkTcIntInstall() method. The Link function is used to install a handler for one of the link blocks as opposed to the primary block.

The DMA system shares a single interrupt for all TC interrupts, and the system will call the installed handler when the particular bit in the TC register becomes set. The handler installer requires an Interrupt Binder Object (See " Interrupt Binder Templates " on page 54.) as an argument to associate a handler function or method and argument for the interrupt forwarding mechanism of Pismo. A second pair of functions, TcIntDeinstall() and LinkTcIntDeinstall() removes any installed handler for the TC bit used by the block.

Once installed, TC interrupts for the entire transfer may be enabled or disabled by a call to TcIntEnable().

The following example shows a full Edma transfer with TC interrupt handling. In this example a class member function is bound to handle the interrupt response.

```
class DmaIsr
{
public:
    typedef void (*IntFtnType)(void * fallow);

    DmaIsr()
        : Binder(*this, &DmaIsr::MyHandler, NULL)  {  }

    void MyHandler(void * fallow)
    {
        qdma_not_done = false;
    }

    ClassMemberHandler<DmaIsr, void *> Binder;
};

DmaIsr Isr;

void EdmaTest()
{
    Edma Ed;

    Ed.Settings().Priority(DmaSettings::priHigh).ElementSize(DmaSettings::is32bit);
    Ed.Settings().ElementIndex(1).ElementCount(50).FrameIndex(1).FrameCount(0);
    Ed.Settings().TCInt(true).TCCode(1).FrameSync(true);
    Ed.Settings().SourceAddr(int(&src_array[0])).SourceIncr(DmaSettings::Incr);
    Ed.Settings().DestinationAddr(dest_array).DestinationIncr(DmaSettings::Incr);
    //
    //  Define a linked
    DmaSettings Cfg;

    Cfg.Priority(1).ElementSize(0).SourceIncr(1).DestinationIncr(1);
    Cfg.TCInt(true).TCCode(1).FrameSync(true);
    Cfg.SourceAddr((int)&src_array[0]).DestinationAddr((int)(dest_array+50));
    Cfg.ElementCount(50).ElementIndex(1);
    Cfg.FrameCount(0).FrameIndex(1);

    Ed.AddLink(Cfg);
    Ed.LinkTcIntInstall( 0, Isr.Binder );
```

```
    Ed.TcIntClear();

    // This EDMA operation will trip a terminal count interrupt when
    // all data has been moved.
    InitArrays();
    Ed.TcIntEnable(true);
    qdma_not_done = true;
    Ed.Submit();
    // We software-initiate the EDMA here, but if this EDMA were using EINT4..7,
    // then an external int hardware pulse would remove need for Ed.Set, below
    Ed.Set();
    while (qdma_not_done)
        ;
    // Need to sync L2 cache with the of SDRAM, so that CPU can see the data
    CACHE_clean(CACHE_L2, dest_array, sizeof(dest_array));
    //
    //  Transfer the second transfer block...
    Ed.Set();
    while (qdma_not_done)
        ;
    // Need to sync L2 cache with the of SDRAM, so that CPU can see the data
    CACHE_clean(CACHE_L2, dest_array, sizeof(dest_array));
}
```

The above example sets up a two block linked transfer triggered by software. A TC Interrupt is configured to signal the completion of each block in the transfer. The mainline waits for each block transfer to finish, as notified by the interrupt handler. Then the next block transfer is triggered by a second call to Set(). The Cache functions are required to assure that the cache and memory contents are back in synchronization.

Linked and Chained blocks

EDMA transfers may span multiple transfer blocks. On the completion of the primary transfer, the first link block is loaded into the primary block and initiated. When this block completes, the next linked block is loaded, and so on. A link block can form a loop, but it is important to remember that the primary block can never be part of a loop. Since it is overwritten by the first linked transfer, this transfer can only occur once. Because of this to make a loop of two transfers requires three blocks to be configured. The primary block contains the first transfer, the first link the second transfer, and the third is a repeat of the first transfer that is linked back to the first link block.

Link blocks are allocated by a call to AddLink(). This call automatically configures the preceding block to link to this newly added block. It returns the index of the newly added block that can be used in order to configure the link block. To form a closed loop in a block chain, call LinkBackTo(). This connects the final block in the chain back to the block whose index is given in the argument.

Transfer chaining is a mechanism for having a transfer trigger another on completion. The ChainTo() and ChainEnable() methods set up a chaining relation between two transfers. Note that on the TI C671x processor, the second transfer must be configured on channels 8-11.

Class EdmaMaster

This class acts as a holder for functions and information common to all EDMA interrupts instead of associated with a single EDMA channel. Only one instance of EdmaMaster is created at program initialization. It is accessed by calling the static member function EdmaMaster::Object().

EdmaMaster contains several functions dealing with the EDMA PRAM. This is a memory region shared among all EDMA objects giving a common storage for configuration blocks. This is a limited resource, so be wary of allocating many Edma blocks and not releasing them. The method ClearPram() clears all the PRAM blocks in a single operation.

EdmaMaster contains several functions dealing with the EDMA PRAM. This is a memory region shared among all EDMA objects giving a common storage for configuration blocks. This is a limited resource, so be wary of allocating many Edma blocks and not releasing them. Also available are functions to give access to the area at the end of the PRAM that is not used by the system. This scratchpad memory might be of use as a shared memory pool in an application.

SBC6713e Example Programs

Under **Sbc6713e\Examples** in the install directory, the baseboard's example programs are installed. Some examples have no host component, and some use the terminal emulator applet as the host. Host examples are written in C++ either under Borland C++ Builder or Microsoft MSVC, or both. Target examples are written using CCS 3.x and DSP/BIOS.

Table 12. SBC6713e Example Programs

Example	Host	Target	Illustrates
AEcho	terminal emulator	DSP/BIOS	Use of DSP/BIOS drivers; Analog output driven from analog input
AnalogCapture	terminal emulator	DSP/BIOS	Repeatedly start Analog Input Streaming to capture points. Useful for high rate input
AnalogIn	terminal emulator	DSP/BIOS	Analog capture into DSP memory. Samples sent to host and saved for analysis.
AnalogOut	terminal emulator	DSP/BIOS	Analog waveform playback from pre-calculated buffer in DSP memory.
ASnap	BCB, VC++7, VC8 (.NET)	DSP/BIOS	Full-rate analog capture to memory, then send to host via busmastering. Illustrates data capture and target/host communication with commands and data packets.
CommonCDB	N/A	N/A	Shared CDB file for all examples.
DioData	terminal emulator	DSP/BIOS	Use of the board Digital I/O
Edma	terminal emulator	DSP/BIOS	Use of Pismo Edma and Qdma wrapper classes with installable interrupt handlers.
FftFloat	terminal emulator	DSP/BIOS	Use of floating point FFTs
Files	terminal emulator	DSP/BIOS	Use of C++ Standard I/O library
FirFloat	terminal emulator	DSP/BIOS	Use of FIR filters

Example	Host	Target	Illustrates
Fp	terminal emulator	DSP/BIOS	
FpEcho	terminal emulator	DSP/BIOS	Use of FPDP driver to flow data through loopback connector
Messages	BCB, VC++7	DSP/BIOS	
Servo	terminal emulator	DSP/BIOS	Use of the PISMO Servo class for servo application
Swi	terminal emulator	DSP/BIOS	Use of Pismo SoftInt class for software interrupts.
Timer	terminal emulator	DSP/BIOS	Use of Pismo ClockBase objects for timebase control.
Uart	BCB	DSP/BIOS	

The Next Step: Developing Custom Code

In building custom code for an application, Innovative Integration recommends that you begin with one of the sample programs as an example and extend it to serve the exact needs of the particular job, or at least refer to the examples to see how some functions are done. Since each of the example programs illustrates a basic data acquisition or DSP task integrated into the target hardware, it should be fairly straightforward to find an example which roughly approximates the basic operation of the application. It is recommended that you familiarize yourself with the sample programs provided. The sample programs will provide a skeleton for the fully custom application, and ease a lot of the target integration work by providing hooks into the peripheral libraries and devices themselves.

Communication with the Host

Overview

Some applications for the Sbc6713e baseboard involve communication with the host CPU in some manner. During development, applications are routinely downloaded from the host following a target reset, even if there is no further Host communications afterwards.

Some applications need to interact with a host program during the lifetime of the program. This may vary from a small amount of information to acquiring large amounts of data. Some examples:

- Passing parameters to the program at start time
- Receiving progress information and results from the application.
- Passing updated parameters during the run of the program, such as the frequency and amplitude of a wave to be produced on the target.
- Receiving alert information from the target.
- Receiving snapshots of data from the target.
- Sending a sample waveform to be generated to the target.

These different requirements require different levels of support to efficiently accomplish. The simplest method supported is a mapping of Standard C++ I/O to the terminal emulator applet that allows console-type I/O on the host. This allows simple data input and control and the sending of text strings to the user.

The next level of support is given by the Packetized Message Interface. This allows more complicated medium-rate transfer of commands and information between the host and target. It requires some software support on the host, unlike the use of standard I/O.

Packetized Message Interface

The DSP and Host are interconnected via a 10BASE-T/100BASE-T Ethernet communications link for the interchange of commands and information. The Pismo libraries provide a packet-based message system between the target and host software. These packets can provide a simple yet powerful means of sending commands and information across the link.

Communication between Host PC and DSP Co-Processor.

Even though the SBC6713e uses an Ethernet link, the 'C6713 doesn't get involved in any TCP/IP protocol related work; this is handled by a dedicated co-processor.

Brief details of the communication protocol: There are basically two aspects of the communication protocol, the host and the target co-processor. The host libraries (Malibu libraries) use a common domain software called the ACE toolkit; a good source of information can be found in:

[1] C++ Network Programming, Mastering Complexity with ACE and Patterns; Douglas C. Smith, Stephen D. Huston; Addison Wesley. The ACE toolkit handles all TCP/IP protocol implementations.

There are two packet types:

1) Command packet.

2) Data packet.

These two packet types use the same ACE_SOCK_Stream methods "send_n() and recv_n()" which transmit and receive data buffers of exactly *n* bytes [1].

We will not explain how the details of the ACE toolkit implementation rather, the structure of the data buffer passed to these methods and the handshake/protocol used between host PC application and Sbc6713e co-processor.

7) Host to Target <u>command</u> packet send_n() / recv_n():

Host (send_n()) ---- [command packet] ----->>Target

Host (recv_n())<<-- [command ack]---------Target

At a high level, the Host will use the following ACE_SOCK_Stream method (please refer to **NetTransferThread_Mb.cpp,** in Malibu library, as a help to understand more of the details, there is also a chapter that goes into the details of host-target communication protocol implementation).

```
client.send_n(MyDataBuff, sizeof(MyDataBuff), &timeout)
```

The previous method will return -1 if the send method timed out. Sending a command from host to target or target to host does not require to send a header packet prior to sending the command. (we will talk about header packets later in this section). A *Command packet* is composed of five-32-bit words.

A ***command*** packet has the following structure:

```
Packet[0] = xcSendCommand; // Packet type.
Packet[1] = CommandId;     // Command ID
Packet[2] = Arg0;          // User defined arguments.
Packet[3] = Arg1;
Packet[4] = Arg2;
```

```
enum XfrCmdTypes { xcSend = 0xB0, xcSendCommand, xcSendData, xcClose, xcSendDataData };
```

After Host has sent the command packet, it will post a receive with the following ACE_SOCK_Stream method:

```
int ackBuff[3];

client.recv_n(ackBuff, sizeof(ackBuff), &timeout);
```

Acknowledgment packets are Three 32-bit words. This is the information one should expect from the Target side:

```
ackBuff[0] = CommStatus; // see Sbc6713e_Mb.h for these codes.

ackBuff[1] = count, number of previously received bytes.

ackBuff[2] = count, number of previously received bytes.
```

When Host received the command acknowledgment, command transfer is understood to be completed.

Note: Even though the command sent from Host to Target DSP is five-32-bit words, the buffer containing the command that is sent from the Co-Processor to the Target DSP is only Four-32-bit words. The packet type (in this case xcSendCommand) is removed from the data buffer and a special method is used to relay this buffer to the Target DSP.

There is currently a restriction on the number of 32-bit-words the inter-processor communication Fifo can handle, and that is -the amount of 32-bit-words in a packet has to be an even number.

8) Target to Host command packet send_n() / recv_n():

Host (recv_n())<<-- [command packet]---------Target

Host (send_n()) ---- [command ack] ----->>Target

The packet structure for command packet and ACK packet are the same as previously discussed.

9) Host to Target data packet send_n() / recv_n():

Host (send_n()) ---- [packet header] ----->>Target

Host (recv_n())<<-- [command ack]---------Target

Host (send_n()) ---- [packet data] ------>>Target

Host (recv_n())<<-- [command ack]---------Target

The ACE_SOCK_Stream methods to receive and send are the same as the ones used when sending a command.

The packet header buffer is an array of five-32-bit words, and has the following structure:

```
Packet[0] = xcSend;         // Packet type.
Packet[1] = Buffer size;    // Size in bytes.
```

```
Packet[2] = 0;
Packet[3] = 0;
Packet[4] = 0;
```

The ACK packet is the same as with the command ack packets.

The second time the send_n() method is invoked, one will use a buffer with raw data, this is the user data to send across to the Target DSP. The requirements of this buffer are the following:

10) Needs to contain at least two-32-bit words. The driver currently does not handle buffers smaller than two-32-bit words (i.e. three bytes, etc.)

11) The number of of 32-bit-words in your data buffer needs to be a multiple of two (i.e. even number of 32-bit words).

12) Target to Host <u>data</u> packet send_n() / recv_n():

Just as with the Command receive operation, the Host will first post a receive using recv_n() ACE_SOK_Stream method, then it will decode the packet type (i.e. xcSend or xcSendCommand, after sending and ack packet back to Target DSP, it will post another receive using recv_n() with the information gotten from the header (i.e. it will pass the number of bytes is expecting to receive), Host needs to send packet acknowledgments for every packet received, this also includes the one for a Header packet.

Communication between Host PC and DSP Target Processor.

The following section describes how to communicate between a Host PC and the C6713 from the perspective of the C6713 DSP. We previously discussed communications between Host and DSP Co-Processor.

The SBC6713e is capable of handling two types of data packets.

1. User data packets.
2. Command data packets.

There is a different C++ method to handle the transmission of these two types of message:

```
1.  Transfer::Send( Buffer & buffer )

2.  Transfer::SendCommand( short CommandId, int Arg0, int Arg1, int Arg2 )
```

In the SendCommand() method, The CommandId must be a number less than 49152 (0xC000), a CommandId with a higher number will conflict with IDs reserved for the system.

There is one method used to receive data. This method makes use multitasking BIOS routines.

```
Transfer::Recv( RecvType & which, Buffer & Buffer )
```

When the above function returns, a command or a message packet may have been received. The call set the which parameter with the type that did arrive -- command or packet. Commands are a four word block with several fields. To simplify the

interpretation of this command the TransferPacketHeader class can be used to isolate the fields of the message. In the code below the Hdr is a transfer packet header. The data is loaded into it from the buffer Buf, and the data fields extracted for use, here by the print function.

```
Hdr.LoadFrom( Buf );
printf("Cmd %d: (%d, %d, %d)\n", Hdr.Type(), Hdr.ArgO(), Hdr.Arg1(),
                                 Hdr.Arg2() );
```

The following piece of code illustrates the three Transfer methods used to communicate with the Host PC.

```
//=============================================================================
//  MsgRecvThread()  --
//=============================================================================

class MsgRecvThread : Thread( priority ) {}
{
   public:
               // Methods.
         void Execute()
         {
            while( !Terminated )
      {
            Transfer              Xfer;
            IntBuffer             Buf;
            TransferPacketHeader  Hdr( new NullTHLIntf( &Hdr ));
            Transfer::RecvType    which;

            Xfr.Recv( which, Buf );            // Wait for message to arrive from
                                               // host. Will block thread until
                                                      // a good message is received received.

            if( which == Transfer::rtCommand )
            {
               Hdr.LoadFrom( Buf );
               printf("Cmd %d: (%d, %d, %d)\n", Hdr.Type(), Hdr.ArgO(), Hdr.Arg1(),
                                                             Hdr.Arg2() );
            }
            else
            {
               int   *Data = Buf.IntAddr();
               printf("PktSz %d: (%x, %x, %x)\n", Buf.Ints(), Data[O], Data[1],
                                                            Data[2] );
            bool   Ok( true );
            for( int i = 1; i < Buf.Ints(); i++ )
               if( Data[i] != 0xEEEE0000 + i )
                  Ok = false;

            if( Ok )
               printf( "Ok\n" );
            else
               printf( "Ok = false" );
            }
      }
   }
};
MsgRecvThread   ReceiveLoop( tpNormal );


//=============================================================================
//  MsgSendThread()  --
```

```
//============================================================================

class MsgSendThread : Thread( priority ) {}
{
   public:
                // Methods.
         void Execute()
         {
            int   cCount( 0 );
            int   Toggle( true );

            while( !Terminated )
            {
               Transfer      Xfr;
               IntBuffer     Buf( 1000 );
               int           *Data = Buf.IntAdder();

               for( int i = 1; i < Buf.Ints(); i++ )
                  Data[ i ] = 0xCCCC0000 + i;

               Sleep( 500 );

               // Send Command or User message.
               if( Toggle )
               {
                  Xfr.SendCommand(0x009, cCount++, 0xDEED, 0xABBA );
                       printf("SendCmd %d: (%d, %d, %d)\n", 0x09, cCount-1,0,0);
                   }
                   else
                   {
                      Data[ 0 ] = cCount++;
                      Xfr.Send( Buf );
                      printf("SendPkt %d: (%d, %d, %d)\n", Buf.Ints(), cCount -1,
                                                                    0, 0);
                   }
                Toggle != Toggle;
             }
          }
       };
MsgSendThread   SendLoop( tpNormal );

//============================================================================
//  Main program  --
//============================================================================
void IIMain()
{
   RecvLoop.Resume();
   SendLoop.Resume();

   Sleep( SYS_FOREVER );
}
```

Details on methods and data members are found in the online help.

C++ Terminal I/O

The terminal emulator is a Host PC application which provides a C++ language-compatible, terminal emulation facility for interacting with the TermIo Pismo library running on an Innovative Integration Matador-family DSP processor.

Using the terminal emulator, it is possible to develop and debug target DSP code while deferring development in Host application code. By using simple, streaming I/O functions within a target application during development, DSP algorithms can be developed independently from Host applications. Later, when a custom Host application code is written, the DSP standard I/O functions may be deleted from the target application and the target application will no longer be dependent on the terminal emulator or the target TermIo libraries. Note that it is currently not possible to use the terminal emulator simultaneously with a custom Host application.

Streaming methods such as << and >> are dispatched by the TermIo object to route text and data between the DSP target and the Host the terminal emulator terminal emulator applet. Text strings are presented to the user via a terminal emulation window and host key-board input data is transmitted back to the DSP. The terminal emulator works almost identically to console-mode terminals common in DOS and Unix systems and provides an excellent means of accessing target program data or providing a simple user interface to control target application operation.

Two different implementations of the terminal emulator exist. The UniTerminal implementation utilizes the communications bus in order to effect character-level data traffic between the Host and the Target DSP. Alternately, the RtdxTerminal implementation uses the JTAG RTDX bus to accomplish this task. Only one of these implementations is provided for each DSP board. Check the Program Files\Innovative\ subdirectory to determine which of these tools is provided with your Dev Kit package.

Target Software

All of the features of the terminal emulator are accessed through the two classes TermIo and TermFile. TermIo provides the basic streaming interface which allows text messages to be formatted and streamed out to the terminal emulator as well streaming in strings and numeric values from the terminal emulator for consumption by target application code. The TermFile class provides a mechanism allowing target applications to open host disk files, perform read and write accesses, and subsequently close these files. See the Files.cpp example for illustrative usage of each of these classes and their functions.

Tutorial

Using the terminal emulator during target software development is simple. The first step is to instantiate a TermIo object through which all terminal I/O will be performed. This occurs implicitly within the cio object as a create-on-first-use behavior.

Use the methods within the TermIo class to format text strings and then stream them to the the terminal emulator applet

```
cio << bold << "7Demonstrate file I/Onn" << normal << endl;
```

Note the use of manipulators, such as bold and normal, to force formatting of the text string as it is streamed to the host. TermIo features many such manipulators to perform functions such as setting text color (setcolor), clearing to end-of-line

(clreol), clearing the screen (cls) and so forth. Other manipulators are available to format numeric values as they are streamed to the host. For example, the phrase

```
term << "Hello" << hex << showbase << 4660 << dec;
```

displays the string "Hello 0x1234" on the console display, converting the integer value 4660 as a hexadecimal number on the target, prior to streaming it to the host. Other manipulators are available providing extensive control over the display of floating point numbers as well as integer values.

It is also frequently necessary to obtain input from an operator during the run-time execution of a target application. For example, it may be necessary to prompt for a sample rate at which analog I/O is to be streamed. The code fragment below illustrates the necessary technique

```
// Prompt the user
term << " Enter a float: " << flush;
float x2;
// Eat user input
term >> x2;
```

The stream manipulator >> is overloaded to allow streaming directly into floating point, integer and string variables directly from the terminal emulator.

To perform file input and output from within target applications, first instantiate a TermFile object, as below

```
TermFile File;
```

Then, use the TermFile Open method to open the file for access on the host using the desired open attributes

```
if (!File.Open("wave.bin", "w+b"))
    {
    term << "nOutput file open error - Program terminating!" << endl;
    term.monitor();
    }
```

This method returns a Boolean indicating success if the file open is successful. To store data into the file or retrieve data from the file, use the Write or Read methods, respectively. For example

```
transferred = File.Write((char*)&Buffer[0], 10000);
```

writes 10000 bytes of Buffer into the disk file. When disk operations have been completed, the file should be closed using the TermFile::Close method.

Detailed Host-Target Communication

Overview

This chapter explains protocol details on how the Host-Transfer communication is performed, thus being a guide for those wanting to develop custom host libraries. You may want to have the Malibu files "NetTransfer_Mb.cpp and Sbc6713e_Mb.cpp files opened. The documentation in this chapter is according to the client firmware version 3.8 in your Sbc6713e target DSP board. The name "client.bin" is misused since the DSP board is really the server.

Clarification: The Sbc6713e acts as a Server; the Malibu library (Host PC software) behaves as a Client.

We use an ACE_SOCK_Stream object called client to open communication sockets, close sockets, and to send and receive data. Please refer to "The ACE Programmers Guide; Huston, Johnson, Syyid; Addison Wesley" for questions regarding ACE source. The Ace Wrappers are free source.

```
"int"   = 32-bit word.
"short" = 16-bit word.
"byte"  = 8-bit word.
```

I. COFF file download.

This procedure downloads an executable file to the target DSP board. It uses socket number 1007. NetCoffLoadThread_Mb.cpp and Sbc6713e_Mb.cpp can be used as references to understand the following explanation.

Protocol:

After establishing and opening the socket with the target board. the Host will do the following:

1. Send a pkDownloadRequest command to target. The data buffer looks like the following:

```
int packet[3];
packet[0] = pkDownloadRequest;
packet[1] = 0;
packet[2] = 0;
```

This is a 4-word (32-bit words) buffer; pkDownloadRequest = 0xA0.

2. Wait for an ACK command from Target. The ACK command is a 3-word buffer:

```
buf[0] = pkDownloadRequestAck.
buf[1] = Size in bytes expected.
buf[2] = byte count received from host.
```

3. Host will parse the CoffFile and send it in chunks of 16384 words, for each packet it expects an ACK command from the Target. Each Host to Target transmission is composed of a header packet and a data packet. Header Packet looks as following:

```
Header[0] = pkSectionHeader;
Header[1] = PacketAddr;
Header[2] = PacketSize;
```

The Data Packet contains the opcodes being downloaded to target. Please read CoffFile_Mb.cpp to understand how the data file is parsed.

When EOF condition is met, the Host will send a final command to start the program on the Target.

```
packet[0] = pkDownloadDone;
packet[1] = 0;
packet[2] = 0;
```

II. Sending data to Target.

It is important to have a little background on how the two DSPs that are in the target Sbc6713e communicate. Basically there is a piece of logic in between the two DSPs, we called it the "LAN FPGA". The function of this FPGA is to serve as a data FIFO between the two DSPs. So whenever the Host PC wishes to send data to the C6713, it will first be processed by the DM642 and then it will be moved to the FIFO; the FIFO is only 256 words deep, so the DM642 CoProcessor will wait until the C6713 reads the contents of the FIFO. If the C6713 does not read data from the FIFO (i.e. because its application code does not do so), then the DM642 will block on this "Send" and thus will not be able to process more data on such socket number. The DM642 software architecture is Multi-threaded, so other aspects of its software will still run. The same process happens when the C6713 wants to send data to the Host PC, if the host PC does not reads it, then the C6713 eventually will block on a "Send"

Another important aspect, if you choose to use Malibu, it is important that before closing your application, you send a notification to the C6713 that you are closing and that you expect your "last" command back, this way if you have a pending "Receive" on a Thread you created, your thread "receive" will be satisfied with this "last command" and thus your application will close normally.

References on the following explanation can be found in NetTransferThread_Mb.cpp and Sbc6713e_Mb.cpp. This is the only method available to send information (any kind) to the Target TMS320C6713 DSP. All other data is processed by the Target DM642 CoProcessor. The procedure is as follows:

1. Host computer opens port number **1008**. This includes configuring the correct type of address (INET), creating a Connector object, and creating a Stream object. This is a TCP/IP connection oriented protocol.

2. In the Malibu libraries, we create a Thread object that will "Wait" for application code to request a data packet or command packet to be sent; if there is no data to send and 45 seconds have elapsed, the Host sends a close command and waits for an ACK sent by Target. If you are writing your own "library", you may or may not send this packet to the target DSP; this "close" command will be processed by the DM642 and will not reach the C6713 DSP:

```
Close packet sent from Host to Target.
  int ClosePacket[5];
```

```
ClosePacket[0] = xcClose;
ClosePacket[1] = 0;
ClosePacket[2] = 0;
ClosePacket[3] = 0;
ClosePacket[4] = 0;

ACK packet sent from target to host. If the DM642 is on a waiting for the
C6713 to read "pending data"
int buf[3];
buf[0] = xcClose;
buf[1] = 0.
buf[2] = 0.
```

3. Now, the real process begins here. To begin, send packet header to target:

```
int Packet[5];
Packet[0] = xcSend, or xcSendCommand, or xcClose;
Packet[1] = Size of Data Buffer in bytes, OR CommandID  **
Packet[2] = 0;
Packet[3] = 0;
Packet[4] = 0;

**CommandID in case of xcSendCommand, Size in case of xcSend, "0" in case of
xcClose. A CommandID is your own application enum which  will be interpreted
in your C6713 DSP application.

ACK packet sent from target to host.
int buf[3];
buf[0] = CommStatusCode;
buf[1] = Received Byte count; // This is 20 bytes (5 words).
buf[2] = Received Byte count;
```

If the command sent from Host was xcSendCommand or xcClose, then the operation completes with the ACK packet sent from Target to Host PC. If the command sent from Host to Target (in the packet header) was xcSend, then the next applies.

4. Host sends a Packet Header to Target:

```
int Header[3];
Header[0] = xcSendData;
Header[1] = Size of Data Buffer to follow, in bytes.
Header[2] = 0;
```

Target Sends ACK to Host PC.

```
ACK packet sent from target to host.
int buf[3];
buf[0] = CommStatusCode;
```

```
buf[1] = Received Byte count; // This is 12 bytes (3 words).
buf[2] = Received Byte count;
```

5. Host PC sends data packet to Target, it does not require additional formatting, this is your own application data and the complete contents of the data is intended to reach the C6713 DSP. This packet has even number of 32-bit words otherwise the inter-processor FIFO is not able to process it. Also, you already specified the amount of bytes you were sending in the previous header packet, if you do not follow this, then there is a great chance of confusion between the client and server programs. Target replies with an ACK.

```
int buf[3];
buf[0] = CommStatusCode;
buf[1] = Received Byte count;
buf[2] = Received Byte count;
```

***** All Data Packets intended to reach the Target C6713 must have an even size of 32-bit words and a minimum of 4 words.**

enum XfrCmdTypes{ xcSend = 0xB0, xcSendCommand, xcSendData, xcClose, xcSendDataData };

enum CommStatusCodes{NetworkTimeoutErr = 0x0, Success, TargetTimeoutErr, TargetErr, UnknownErr, CommAborted};

xcSendDataData is no longer used.

It is important to understand that this operation is Host-initiated, so the DM642 DSP will wait first on Host activity, then it will forward the data to the C6713 DSP, the C6713 DSP program must be expecting to receive this data, otherwise, Host PC will timeout within 5 seconds of no response (that is in the case of Malibu).

II. Receiving data from Target.

The port number used from this operation is **1009**. This is the only port the Target DSP (C6713) is able to send data to Host. There are other ports/sockets used where the Target Co-Processor (DM642) sends data to Host. As a reference please have the file NetTargetTransfer_Mb.cpp opened.

The concept: Your Host code sends a “request data” command to the DM642 CoProcessor, the DM642 in turn goes to its FIFO (see explanation on the beginning of this chapter) and “blocks” on data from the C6713; the Host code (in our implementation) has a Timeout feature, thus if the C6713 has no data to send it will generate a Timeout event to inform the host application code that no data has been received; the host code may opt to “halt” the pending receive request or ignore the event and keep on waiting, in our implementation we keep on posting “network receives” with 5 second timeouts until data is received or until the application code requests a “Halt”.

1. Host opens / connects to port **1009**. This includes configuring the correct type of address (INET), creating a Connector object, and creating a Stream object. This is a TCP/IP connection oriented protocol.

2. Host sends data request to Target.

```
int xReply[3];
xReply[0] = xcRecvPosted;
xReply[1] = 0;
xReply[2] = 0;
```

3. Target Co-Processor receives this request and "blocks" on data to be received to C6713. When the C6713 has data to send to Host, the DM642 "un-block" requirement will be satisfied and a Data Packet Header OR Command Packet will be sent to Host.

```
Command Packet.
int XfrPacket[5]
XfrPacket[0] = pkRecvCommandRequest;
XfrPacket[1] = CmdBuf[0]; // CmdBuf received from C6713.
XfrPacket[2] = CmdBuf[1];
XfrPacket[3] = CmdBuf[2];
XfrPacket[4] = CmdBuf[3];

OR

Data Packet Header.
int XfrPacket[5]
XfrPacket[0] = pkRecvPacketRequest;
XfrPacket[1] = Buf.Bytes(); // Number of bytes to follow. In the next
XfrPacket[2] = N/A;         // transaction, Host must get ready to
XfrPacket[3] = N/A;         // receive this next data.
XfrPacket[4] = N/A;
```

If your Host code does not receive any data once you have requested it, and supposing that nothing has gone wrong with the hardware, then you may implement an "application level protocol" that is you send commands to the C6713 (read previous section "Send Data to Target") to request status, then using this socket/port you should receive your DSP status, if you still receive no data, then either your application protocol is not implemented correctly or the hardware is in a bad state. You may choose to send a reset command (using the reset command port/socket number) to the DM642 and try to reconnect few seconds later.

4. Host sends ACK packet to Target.

```
Reply[0] = pkRecvRequestAck;
Reply[1] = 0;
Reply[2] = 0;
```

5. If Target previously sent a Command Packet, then operation ends here. DM642 will close its socket if Host closed its socket. Dm642 will then go into a "wait" state, and it will stay there until Host re-establishes communications.

6. If Target had sent a Data Packet Header, then Target now sends its Data Buffer to Host. Host must be prepared to receive this larger buffer. C6713 should only send even number of 32-bit words to Host PC since the Inter-Processor FIFO cannot deal with odd number of 32-bit words and also, internally the DSPs have and algorithm to to send data as bursts which only deals with even number of words and at a minimum 4 words.

7. Host then sends an ACK packet to Target

```
Reply[0] = pkRecvDataAck;
Reply[1] = 0;
Reply[2] = 0;


enum PakTypes{pkRecvCommandRequest= 0xC0,
    pkRecvPacketRequest,
    pkRecvRequestAck,
    pkRecvData,
    pkRecvDataAck
    };
```

III. Sending RESET command to Target.

1. Open socket number 1023. Send Command to Target:

```
int packet[3];
packet[0] = pkMasterReset;
packet[1] = 0;
packet[2] = 0;
```

2. Target sends back ACK packet:

```
int aSendPacket[3];
aSendPacket[0] = pkMasterResetAck;
aSendPacket[1] = 0;
aSendPacket[2] = 0;
```

3. Host closes all communications with Target.

```
enum PakTypes{pkFirmwareInfoRequest = 0x50, pkBootAppInfoRequest,
    pkNetInfoRequest, pkSetBootApp, pkWriteNetInfo,
    pkInfoRequestAck, pkWriteBdRev, pkMasterReset,
    pkMasterResetAck, NAK
    };
```

IV. Requesting Board information.

1. Host opens socket number 1021, and sends the following command to Target:

```
int packet[3];
packet[0] = pkFirmwareInfoRequest;
packet[1] = 0;
packet[2] = 0;
```

2. Target then sends the following information and is subject to change.

```
int aSendPacket[200];
aSendPacket[0] = pkInfoRequestAck;
aSendPacket[1] =  Interface logic version.
aSendPacket[2] = Lan logic version.
aSendPacket[3] = Client version (running on DM642 Co-Processor).
aSendPacket[4] = Board serial number.
aSendPacket[5] = Board version.
aSendPacket[6] = Static IP or DHCP usage (bool).
From here on, data is sent as null terminated strings.
aSendPacket[7] = Static IP address, even if using DHCP, we send the saved
static address.
aSendPacket[11] = Mask.
aSendPacket[15] = Gateway address.
aSendPacket[19] = DNS.
aSendPacket[23] = Domain name.
aSendPacket[199] = string termination. '\0'
```

3. Host closes socket.

V. Write Board information.

This is used to write certain information to the Target board like IP information and to boot Innovative's "Benign" program or a custom program stored in flash. You can also write the Serial Number of the board and board revision, this operation is not suggested since Serial Number and Board Revision are used to set the MAC address of the board, this command is only for use in-house.

1. Host opens socket number 1022.

2. Host sends a write request packet.

```
int packet[200];    // Data to send
packet[0] = pkWriteNetInfo;  // To set IP addr, mask, etc.
packet[1] = ip addr \0 mask \0 gateway \0 dns\0 domain \0
            use static IP ('Y'/'N') \0.
```

As you can see the information following the packet type is a group of null-terminated strings

OR, the Host may send the following packet:

```
int packet[200];    // Data to send
packet[0] = pkSetBootApp;
packet[1] = Boolean value, true = Boot customer's app from flash.
```

Another supported command (for use In-House only as this causes the MAC address of the board to change).

```
int packet[200];    // Data to send
packet[0] = pkWriteBdRev;
packet[1] = Board revision as ASCII character.
packet[0] = Board serial number as short.
```

3. Target then sends following ACK packet:

```
int aSendPacket[3];
aSendPacket[0] = pkInfoRequestAck;
aSendPacket[1] = 0;
aSendPacket[2] = 0;
```

VI. Burning Firmware.

This command is used to load new firmware (i.e. new interface or lan logic, new client program, new customer's application program).

1. Host opens socket number 1020.
2. Host sends Command Packet as following:

```
int Packet[3];
Packet[0] = Download request type.
Packet[1] = Image size in bytes.
Packet[2] = Image Checksum = sum of contents of all image bytes.
```

3. Target sends ACK packet to Host.

```
int aSendPacket[3];
aSendPacket[0] = pkLogicDownloadRequestAck;
aSendPacket[1] = 0x00;
aSendPacket[2] = 0x00;
```

4. The image file is sent in data buffers of less or equal to 16384 bytes in size. The process is that the Host will send a "Packet Header, then will wait for ACK, then it will send the Data Buffer portion of that header and wait for an ACK:

   a) Host to Target

```
int Header[3];
Header[0] = pkChunkHeader;
Header[1]= 16384 bytes or the remaining of the image file <= 16384
Header[1] = 0;
```

   b) ACK from Target.

```
int aSendPacket[3];
aSendPacket[0] = pkChunkHeaderAck;
aSendPacket[1] = 0x00;
aSendPacket[2] = 0x00;
```

   c) Host sends "Data chunk" and Target ACKs it.

```
aSendPacket[0] = pkChunkDataAck;
aSendPacket[1] = 0x00;
aSendPacket[2] = 0x00;
```

   4. (c) will continue until last chunk of data sent, when this happens, Host will send following command.

```
packet[0] = pkLogicDownloadDone;
packet[1] = 0;
packet[2] = 0;
```

5. Target will acknowledge this command:

```
aSendPacket[0] = pkLogicDownloadDoneAck;
aSendPacket[1] = (int)status = ibOk,ibBadXmitChecksum,ibBadBurnChecksum;
aSendPacket[2] = 0x00;

    enum PakTypes
    {
        pkLanLogicDownloadRequest   = 0xE0,
        pkIntfLogicDownloadRequest,
        pkLogicDownloadRequestAck,
        pkLogicDownloadDone,
        pkLogicDownloadDoneAck,
        pkChunkHeader,
        pkChunkHeaderAck,
        pkChunkDataAck,
        pkLoaderDownloadRequest,
        pkTalkerDownloadRequest,
```

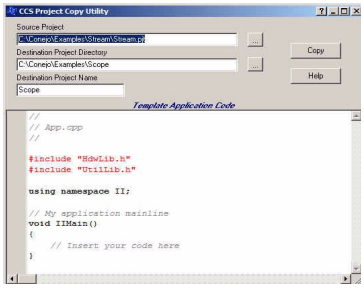
```
    pkAppDownloadRequest
};

enum ImageBurnStatus{ibOk,ibBadXmitChecksum,
                     ibBadBurnChecksum};
```

6. Host closes socket.

   Please refer tp SRecord_Mb.cpp and BinFile_Mb.cpp so that you may see how we parse the different data files, also refer to NetImageLoad_Mb.cpp.

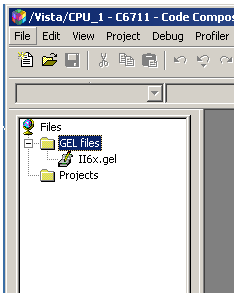
Building a Target DSP Project

Building a project suitable for a Matador or Velocia baseboard requires a particular setup of the project.

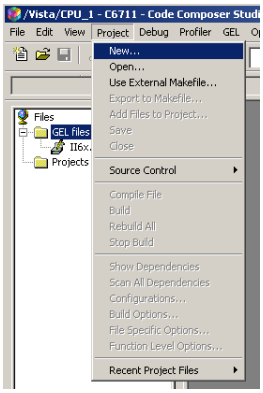


By far, the easiest way to create a new DSP project is by using an existing project as a template. The CopyCcsProject applet provided in the Pismo Toolset automates this task. To use this utility, select an existing Code Composer project as the Source Project, typically one of the example programs supplied in the Pismo Toolset. Next, select the directory into which you wish the new project to be created, using the Destination Project Directory edit control. Then, edit the Destination Project Name for the newly-created project. Finally, click the Copy button to create the new project from the template. The new project may be opened and used within Code Composer.

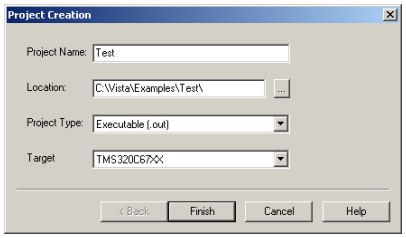
Alternately, you may follow the manual steps below to create a new target DSP project. The project name used below is called Test, but you should name your project appropriately for your application.



Start Code Composer Studio. In the default configuration, the project window will contain no projects but will contain the default Innovative-supplied board initialization GEL file.

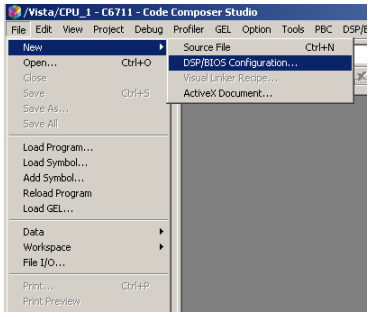


Click Project | New on the menu bar to create a new DSP project.

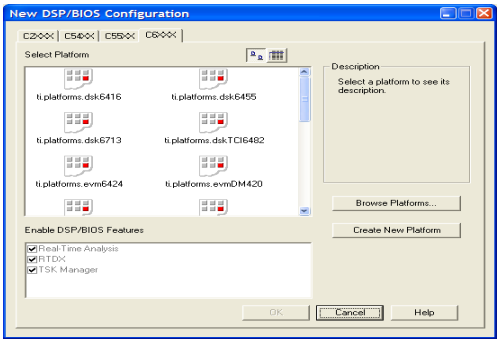


Specify the location for the new project and its name. In this example, a new project called Test is being created in the `Sbc6711-Pismo\\Examples\\` directory. Change the location to accommodate your board type and processor type.

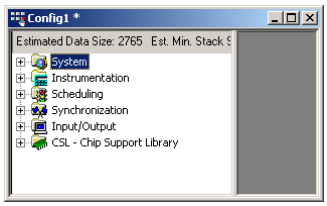
After the new project has been created, it will appear in the CCS project window under the Projects folder.



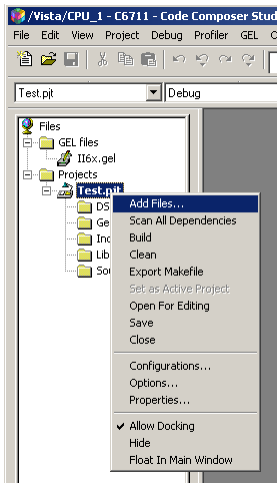
Click File | New | DSP/BIOS Configuration to create a new TCF file for use in the project.



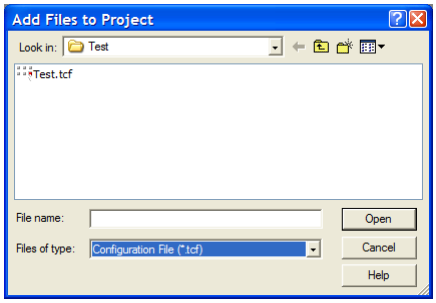
Select the relevant template for the baseboard from the list in the New DSP/BIOS Configuration dialog box.



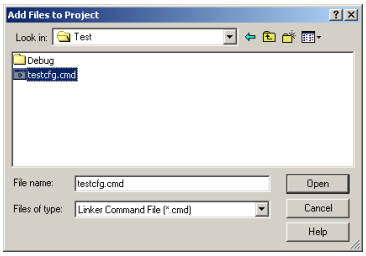
By default, this TCF will be named Configuration1. Save it as `Test.TCF`.



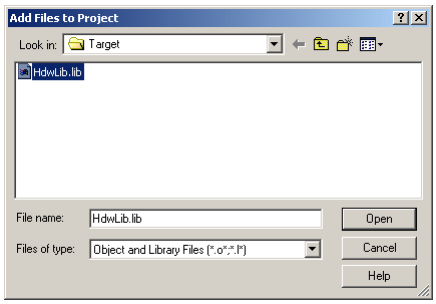
Though the TCF and its support files have been created on disk, you must manually add them to the Test project. Right-click on `Test.pjt` in the Project window to invoke the project hot menu. Click Add Files to add a file to the project.



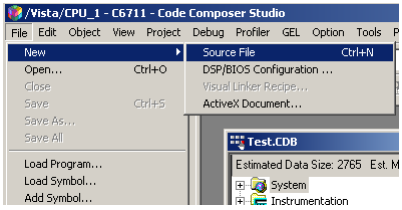
Select the the newly-created `Test.tcf` for addition to the project. This will implicitly add the auto-generated files `Testcfg.s62 (Testcfg.s64 for Velocia cards)` and `Testcfg_c.c` to the project as well.



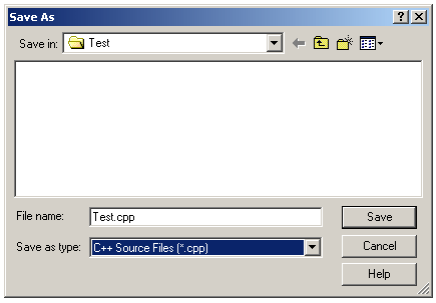
Right-click on `Test.pjt` in the project window, click “Add Files” then select the the newly-created `Test.cmd` for addition to the project.



Right-click on `Test.pjt` in the project window, select Add Files, then browse to the Examples directory and select `Examples.cmd` for addition to the project.



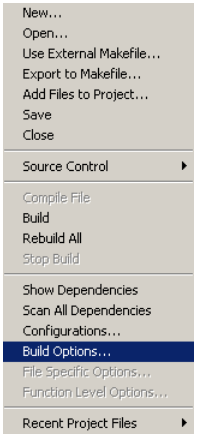
Add an new C++ source file to the project. Click File | New | Source File to create an empty source document.

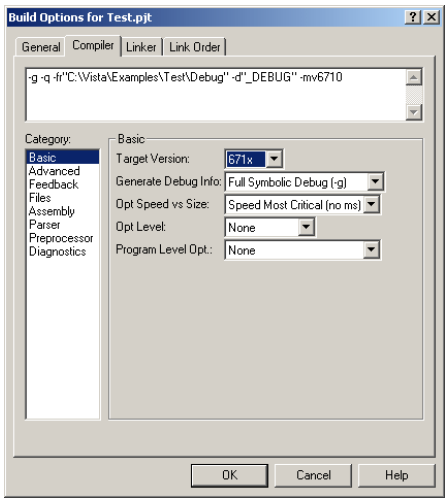


Rename the new source document to `Test.cpp`. To use the Pismo libraries, you must use C++ files and the C++ compiler, even if you intend to restrict your own coding to the C subset of C++

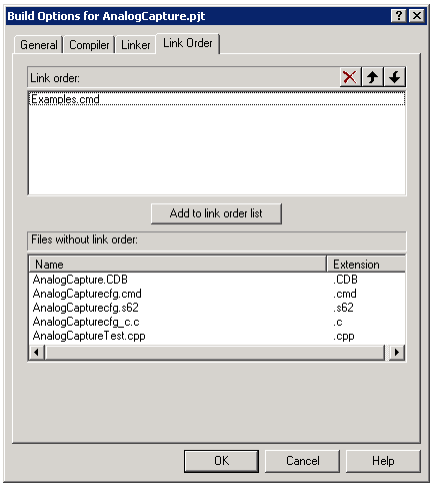
```
//
// App.cpp
//

#include "Pismo.h"

using namespace II;

// Application mainline
void IIMain()
{
    // Insert your code here
}
```

Type the boilerplate code below into your source file. This is the minimum code needed for any Pismo C++ application.

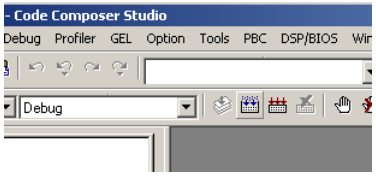




Click the menu Project | Build Options to invoke the compiler Build Options dialog. Then select the Files Category, then enter the pathspec to the `Examples.opt` file in the Examples directory to the Options File edit box.



Click on the Link Order tab, then add `Examples.cmd` to the Link Order List.



Click the Incremental Build button to rebuild the template application. It should compile and link without errors.

Writing a Program

The basic program given in the example above includes a 'Main' function `IIMain()`. DSP/BIOS, the OS used in the Pismo library, uses code inserted after exiting from the normal C-language `main()` to initialize features of DSP/BIOS. This means that some language features are not available then. To avoid these problems the Pismo library provides a `main()` function and uses it to create a single thread. This thread, when executed, calls the `IIMain()` function. Inside of this thread, all DSP/BIOS is initialized and ready for use. It is required that the user include this function, and use it as the equivalent of the old main process function in C.

Host Tools for Target Application Development

The Innovative Integration Pismo Toolset allows users of Innovative DSP processor boards to develop complete executable applications suitable for use on the target platform. The environment suite consists of the TI Optimizing C++ Compiler, Assembler and Linker, the Code Composer debugger and code authoring environment as well as Innovative's custom Windows applets (such as the terminal emulator).

Code Composer Studio is the package used to automate executable build operations within Innovative's Pismo Toolsets, simplifying the edit-compile-test cycle. Source is edited, compiled, and built within Code Composer Studio, then downloaded to the target and tested within either the Code Composer Studio debugger or via the terminal emulator.

Code Composer Studio may be used for both code authoring and code debugging. Details of constructing projects for use on Innovative DSP platforms are given in the above section of this chapter.

Do not confuse the creation of target applications (code running on the target DSP processor) with the creation of host applications (code running on the host platform). The TI tools generate code for the TI DSP processors, and are a separate toolset from that needed to create applications for the host platform (which would consist of some native compiler for the host processor, such as Microsoft's Visual C++ or Borland Builder C++ for IBM compatibles). To create a completely turnkey application with custom target and host software, *two* programs must be written for two separate compilers. While Innovative supports the use of Microsoft C/C++ for generation of host applications under Windows with sample applications and libraries, we do not supply the host tools as part of the Development Environment. For more information on creating host applications, see the section in this manual on host code development.

This section supplies information on the use of the development environment in creating custom or semicustom target DSP software. It is not intended as a primer on the C++ language. For information on C/C++ language basics, consult one of the primer books available at your local bookstore.

Components of Target Code (.cpp, .tcf, .cmd, .pjt)

In general, DSP applications written in TI C++ require at least three files: a .cpp file (or "source" file) containing the C++ source code for the application a .cmd file (or "command" file) which contains the target-specific memory-map and build data needed by the linker, a .tcf file (or "command database" file) which specifies the properties of the BIOS operating system used within the application and a .pjt file ("project" file) which centralizes all project-specific options, settings and files. There may also be one or more .asm assembler source files, if the user has coded any portions of the application in assembly language.

Edit-Compile-Test Cycle using Code Composer Studio

Nearly every computer programming effort can be broken down into a three step cycle commonly known as the edit-compile-test cycle. Each iteration of the cycle involves editing the source (either to create the original code or modify existing code), followed by compiling (which compiles the source and creates, or builds, the executable object file), and finally downloading and testing the result to see if it functions in the desired fashion. In the Innovative Integration development system these stages are accomplished within the Code Composer integrated development environment (IDE).

By using Code Composer Studio, these stages of the programming cycle are accomplished entirely within the IDE. The project features of Code Composer Studio support component file editing and compilation stages, along with allowing the executable result to be downloaded and tested on the target hardware. This fully integrated programmers environment is more user-friendly then the basic command line interface, which comes standard with the TI tools.

Automatic projectfile creation

When a project is created, opened, modified, built or rebuilt, the Code Composer Studio dependency generator automatically generates a project makefile (named `<project file>.pjt`, located in the project directory), which is capable of rebuilding the project's output file from its components.

This file is automatically submitted to the internal make facility whenever you click on build or rebuild within Code Composer Studio. The make facility automatically constructs the output file by recompiling the out-of-date source files including the dependencies contained within those source files.

Rebuilding a Project

It is sometimes necessary to force a complete rebuild of an output file manually, such as when you change optimization levels within a project. To force a project rebuild, select Project | Rebuild All from the Code Composer Studio menu bar.

IIMain replaces main.
Due to restrictions within Dsp/Bios, not all BIOS features may be safely used within `main()`, since it is called early in the system initialization sequence. To circumvent this limitation, Pismo automatically constructs a default thread running within normal priority and starts this thread automatically. The entry point function in this thread is called IIMain, and all Pismo applications must define this function. This function is intended to replace main in your application programs. You may safely call any BIOS function within IIMain.

Running the Target Executable

The test program may be converted into a simple, "Hello World!" example, by using the built-in standard I/O features within Pismo. Bring up the `Test.cpp` source file edit screen. Scroll down the source file by using cursor down button until you reach the `IIMain()` function. Edit it as follows:

```
#include "HdwLib.h"
#include "UtilLib.h"

        cio << init;
cio << "Hello World!" << endl;
```

```
cio.monitor();
```

You can now compile the new version by executing Build from the Project menu (or by clicking on its toolbar icon). This causes Code Composer Studio to start the compiler, which produces an assembly language output. The compiler then automatically starts the assembler, which produces a `.obj` output file (`test.obj`). Code Composer Studio then invokes the TI Linker using the `testcfg.cmd` file, which is located in the project directory. This rebuilds the executable file using the newly revised `test.obj` . If no errors were encountered, this process creates the downloadable COFF file `test.out`, which can be run on the target board. At this point, the program may be run using the terminal emulator applet, which may be invoked using the terminal emulator shortcut located within the target board program group created during the Pismo Libraries installation process. In the terminal emulator, download the `test.out` file. The program runs and outputs the message "Hello, World" to the terminal emulator window.

If errors are encountered in the process, Code Composer Studio detects them and places them in the build output window. If the error occurred in the compiler or assembler (such as a C++ syntax error), the cursor may be moved to the offending line by simply double-clicking on the error line within the build output window, and the error message will be displayed in the Code Composer Studio status bar. If the linker returns a build error, the build output window shows the error file. From this information, the linker failure can be determined and corrected. For example, if a function name in a call is misspelled, the linker will fail to resolve the reference during link time and will error out. This error will be displayed on the screen in the build output window.

Note:
Be sure to start the terminal emulator **BEFORE** starting Code Composer, to avoid resetting the DSP target in the midst of the debugging session. If the terminal emulator is not yet running and you wish to run the Test object file, perform the following steps.

1. Execute Debug | Run Free to logically disconnect the DSP from the debugger software.

2. Terminate the Code Composer Studio application.

3. Invoke the terminal emulator application.

4. Restart the Code Composer Studio application.

This outlines the basics of how to recompile the existing sample programs within the Code Composer Studio environment.

Anatomy of a Target Program

While not providing much in the way of functionality, the `test` program does demonstrate the code sequence necessary to properly initialization the target. The exact coding, however, is very specific to the I.I. C Development Environment, target boards, and is explained in this section in order to acquaint developers with the basic syntax of a typical application program.

```
/*
*       HELLO.CPP
*       Test file/program for target board.
*/

#include "Pismo.h"
```

```
IIMain()
{
        cio << init;

        cio << "Hello World!" << endl;
        cio << "\nEchoing keystrokes..." << endl;

        char key;

        do
                {
                cio >> key;
                cio << key << flush;
                }
        while(key != 0x1b);

        cio.monitor();
}
```

The two lines of the program that being with a "#" are #include statements, which include the header files for the hardware and utility I/O libraries. These include prototypes for all the library classes within Pismo.

The `cio << init` invocation will setup the standard monitor I/O interface and reset the terminal window. The next lines perform the basic standard I/O functions of printing "Hello World!" & "Echoing keystrokes...". These two lines are where custom code could be inserted.

The following do-loop sequence simply echoes keys typed at the terminal emulator back to the terminal display, until the `Esc` key is pressed. When `Esc` is pressed, the `cio.monitor()` function effectively terminates the program, except that interrupts are still active and interrupt handlers (if they had been installed) would still execute properly.

The `test` program is very simple, but it contains the basic components of a typical DSP application, as well as the initialization needed to interact with the terminal emulator.

Use of Library Code

Library routines can be compiled and linked into your custom software simply by making the appropriate call in the source and adding the appropriate library to the linker command file. Refer to the library reference within the Pismo online help for library location information on each class and method.

In general, user software needs to #include the relevant library header file in source code. The header files define prototypes for all library functions as well as definitions for various data structures used by the library functions. The files `HdwLib.h` and `UtilLib.h` should be included within all programs; The file `DspLib.h` should be included if a program uses functions in the DspLib signal processing library.

Example Programs

Under **<baseboard>\Examples** in the install directory, the baseboard's example programs are installed. Some examples have no host component, and some use the terminal emulator applet as the host. Host examples are written in C++ either under

Borland Builder or Microsoft MSVC, or both. Target examples are written using CCS 3.3 and DSP/BIOS. Note that not all of the examples listed below are available for all targets.

Table 13. Pismo Example Programs

Example	Host	Target	Illustrates
FftFix	terminal emulator	DSP/BIOS	Use of Fourier class to perform forward and inverse FFTs
FirFix	terminal emulator	DSP/BIOS	Use of BlockFir class to perform FIR filter functions.
Edma	terminal emulator	DSP/BIOS	Use of Pismo Edma and Qdma wrapper classes with installable interrupt handlers.
Files	terminal emulator	DSP/BIOS	Use of C++ Standard I/O library
CpuInRate	BCB MSVC	DSP/BIOS	Use of Target to Host message and data packet passing via PCI bus.
CpuOutRate	BCB MSVC	DSP/BIOS	Use of Host to Target message and data packet passing via PCI bus.
LinkPort	BCB	DSP/BIOS	Use of LinkPort driver to flow data between all processor in mesh
Swi	terminal emulator	DSP/BIOS	Use of Pismo SoftInt class for software interrupts.
Timer	terminal emulator	DSP/BIOS	Use of Pismo ClockBase objects for timebase control.

The Next Step: Developing Custom Code

In building custom code for an application, Innovative Integration recommends that you begin with one of the sample programs as an example and extend it to serve the exact needs of the particular job. Since each of the example programs illustrates a basic data acquisition or DSP task integrated into the target hardware, it should be fairly straightforward to find an example which roughly approximates the basic operation of the application. It is recommended that you familiarize yourself with the sample programs provided. The sample programs will provide a skeleton for the fully custom application, and ease a lot of the target integration work by providing hooks into the peripheral libraries and devices themselves.

Developing Host Applications

Developing an application will more than likely involve using an integrated development environment (IDE) , also known as an integrated design environment or an integrated debugging environment. This is a type of computer software that assists computer programmers in developing software.

The following sections will aid in the initial set-up of these applications in describing what needs to be set in Project Options or Project Properties.

Borland Turbo C++

BCB10 (Borland Turbo C++) Project Settings

When creating a new application with File, New, VCL Forms Application - C++ Builder

Change the Project Options for the Compiler:

Project Options
++ Compiler (bcc32)
C++ Compatibility
Check 'zero-length empty base class (-Ve)'
Check 'zero-length empty class member functions (-Vx)'

In our example Host Applications, if not checked an access violation will occur when attempting to enter any event function.

i.e.
Access Violation OnLoadMsg.Execute – Load Message Event
Because of statement
Board->OnLoadMsg.SetEvent(this, &ApplicationIo::DoLoadMsg);

Change the Project Options for the Linker:

Project Options
Linker (ilink32)
Linking – **uncheck 'Use Dynamic RTL'**

In our example Host Applications, if not unchecked, this will cause the execution to fail before the Form is constructed.

Error: First chance exception at $xxxxxxxx. Exception class EAccessViolation with message "Access Violation!"
Process ???.exe (nnnn)

Other considerations:

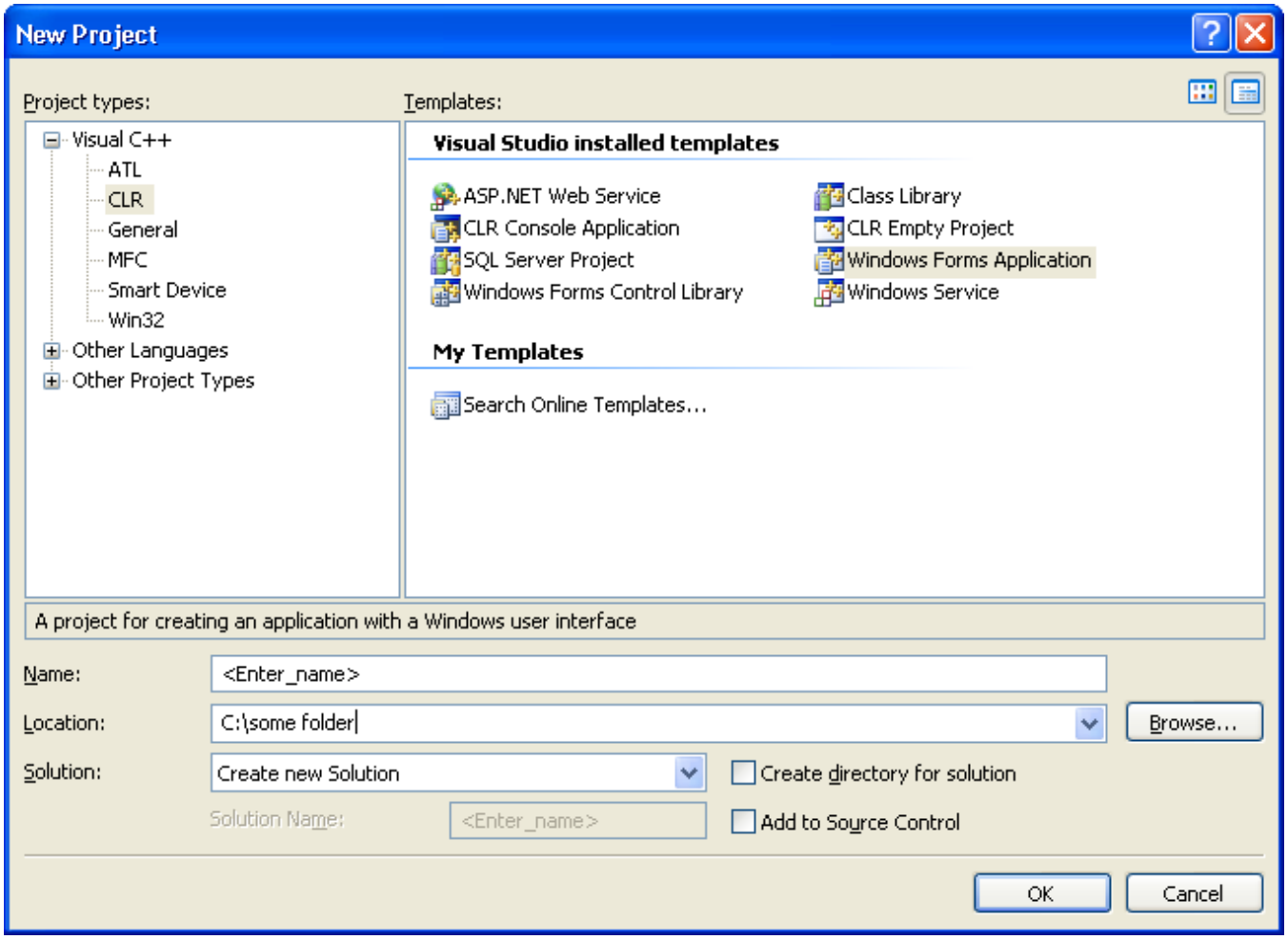
Project Options
++ Compiler (bcc32)
Output Settings
check – Specify output directory for object files(-n)
(release build) Release
(debug build) Debug
Paths and Defines
add Malibu
Pre-compiled headers
uncheck everything
Linker (ilink32)
Output Settings
check – Final output directory
(release build) Release
(debug build) Debug
Paths and Defines
(ensure that Build Configuration is set to All Configurations)
add Lib/Bcb10
(change Build Configuration to Release Build)
add lib\bcb10\release
(change Build Configuration to Debug Build)
add lib\bcb10\debug
(change Build Configuration back to All Configurations)

Packages
uncheck - Build with runtime packages

Microsoft Visual Studio 2005

Microsoft Visual C++ 2005 (version 8) Project Properties

When creating a new application with File, New, Project with Widows Forms Application:



Project Properties (Alt+F7)
Configuration Properties

Project Defaults	
Configuration Type	Application (.exe)
Use of MFC	Use Standard Windows Libraries
Use of ATL	Not Using ATL
Minimize CRT Use in ATL	No
Character Set	Use Unicode Character Set
Common Language Runtime support	Common Language Runtime Support (/clr)
Whole Program Optimization	No Whole Program Optimization

C++
General
Additional Include Directories

Malibu
PlotLab/Include – for graph/scope display

Code Generation
Run Time Library

Multi-threaded Debug DLL (/Mdd)

Precompiled Headers
Create/Use Precompile Headers

Not Using Precompiled Headers

Linker
Additional Library Directories

Innovative\Lib\Vc8

If anything appears to be missing, view any of the example sample code Vc8 projects.

Summary

Developing Host and target applications utilizing Innovative DSP products is straightforward when armed with the appropriate development tools and information.

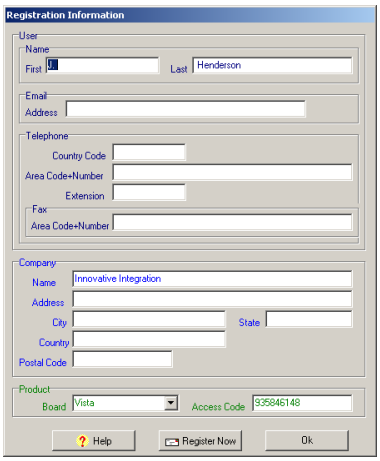
Applets

The software release for a baseboard contains programs in addition to the example projects. These are collectively called "applets". They provide a variety of services ranging from post analysis of acquired data to loading programs and logic to a full replacement host user interface. The applets provided with this release are described in this chapter.

Shortcuts to these utilities are installed in Windows by the installation. To invoke any of these utilities, go to the Start Menu | Programs | <<Baseboard Name>> and double-click the shortcut for the program you are interested in running.

Registration Utility (NewUser.exe)

Some of the Host applets provided in the Developers Package are keyed to allow Innovative to obtain end-user contact information. These utilities allow unrestricted use during a 20 day trial period, after which you are required to register your package with Innovative. After, the trial period operation will be disallowed until the unlock code provided as part of the registration is entered into the applet. After using the NewUser.exe applet to provide Innovative Integration with your registration information, you will receive:

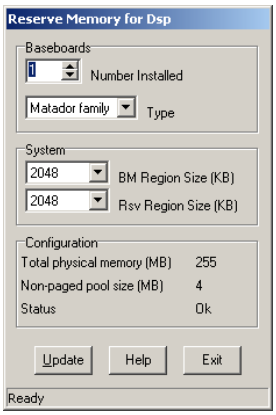


The unlock code necessary for unrestricted use of the Host applets

A WSC (tech-support service code) enabling free software maintenance downloads of development kit software and telephone technical hot line support for a one year period.

Reserve Memory Applet (ReserveMemDsp.exe)

Each Innovative PCI-based DSP baseboard requires 2 to 8 MB of memory to be reserved for its use, depending on the rates of bus-master transfer traffic which each baseboard will generate. Applications operating at transfer rates in excess of 20 MB/sec should reserve additional, contiguous busmaster memory to ensure gap-free data acquisition.

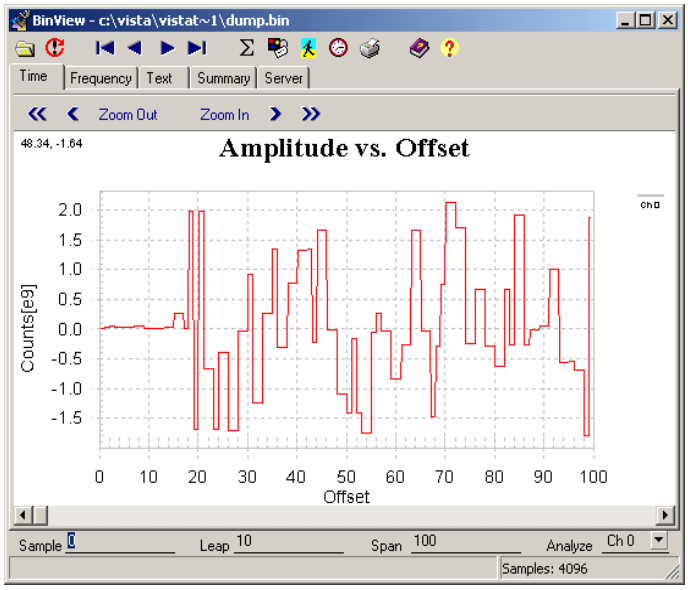


To reserve this memory, the registry must be updated using the ReserveMemDsp applet. If at any time you change the number of or rearrange the baseboards in your system, then you must invoke this applet to reserve the proper space for the busmaster region. See the Help file ReserveMemDsp.hlp, for operational details.

Data Analysis Applets

Binary File Viewer Utility (BinView.exe)

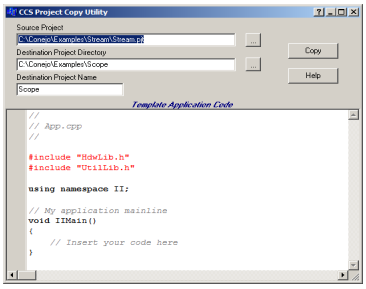
BinView is a data display tool specifically designed to allow simplified viewing of binary data stored in data files or a resident in shared DSP memory. Please see the on-line BinView help file in your Binview installation directory.



Target Programming Applets

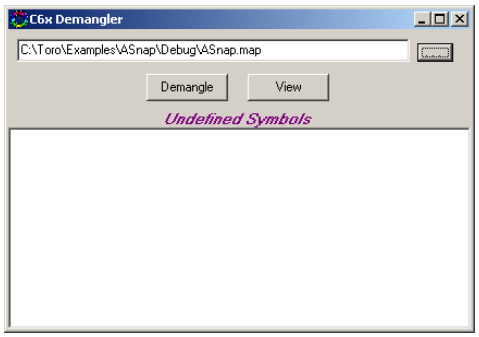
Target Project Copy Utility (CopyCcsProject.exe)

The CopyCcsProject.exe applet is used to copy all project settings from a known-good template project into a new DSP Code Composer project. This simplifies new project development, by eliminating the multi-step process of copying the myriad individual project settings from a source project in a newly-created project.



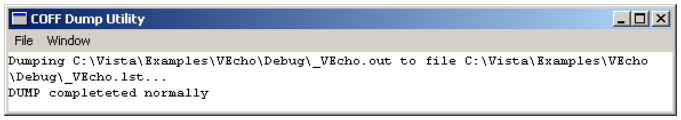
Demangle Utility (Demangle.exe)

The Demangle applet is designed to simplify use of the TI dem6x.exe command-line utility. When building C++ applications, the built-in symbol mangler in the TI compiler renders symbolic names unreadable, such that missing or unresolved symbol errors displayed by the linker no longer correlate to the symbol names within your code. To work around this limitation, enable map file generation within your CCS project. Then, browse to the map file produced by the linker using the Demangle utility. The utility will display proper symbol names for all unresolved externals.

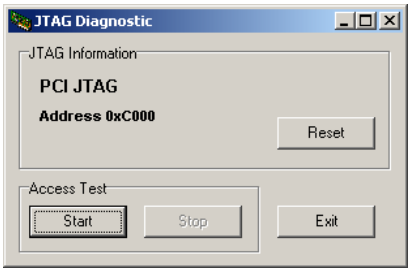


COFF Section Dump Utility (CoffDump.exe)

CoffDump.exe parses through a user-selected COFF file stored on the hard disk and ascertains the complete memory consumption by the DSP program. Memory usage for each of the sections defined in the applications command file are tabularized and the results are written to the Windows NotePad scratch buffer.



JTAG Diagnostic Utility (JtagDiag.exe)



JtagDiag.exe is used to re-initialized the JTAG scan-path interface which connects the Code Hammer debugger's PCI plug-in board with the target DSP. Use this utility prior to invoking Code Composer Studio, to insure that the communications link is viable and clear. This utility is also convenient in confirming that the Code Hammer installation is complete and correct.

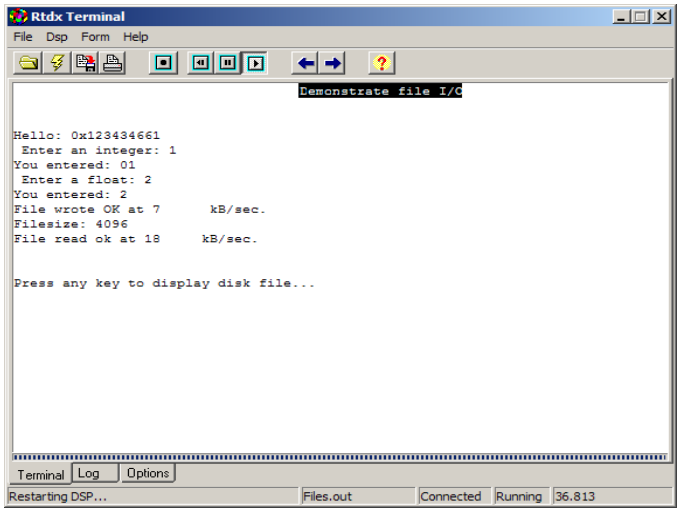
RtdxTerminal - Terminal Emulator

This applet provides a C++ language-compatible, standard I/O terminal emulation facility for interacting with the TermIo library running on an Innovative Integration target DSP processor. Display data is routed between the DSP target and this Host the terminal emulator applet in which ASCII output data is presented to the user via a terminal emulation window and host keyboard input data is transmitted back to the DSP. The terminal emulator works almost identically to console-mode terminals common in DOS and Unix systems, and provides an excellent means of accessing target program data or providing a simple user interface to control target application operation during initial debugging.

RtdxTerminal is implemented as an out-of-process extension to Code Composer Studio. Consequently, it must be used in conjunction with CCS and a JTAG debugger - it cannot operate stand-alone.

The terminal emulator is straightforward to use. The terminal emulator will respond to stdio calls automatically from the target DSP card and should be running before the DSP application is executed in order for the program run to proceed normally. The DSP program execution will be halted automatically at the first stdio library call if the terminal emulator is not executing when the DSP application is run, since standard I/O uses hardware handshaking. The stdio output is automatically printed to the current cursor location (with wraparound and scrolling), and console keyboard input will also be displayed as it is echoed back from the target.

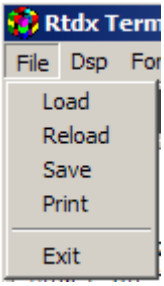
The terminal emulator also supports Windows file I/O using the TermFile library object.



Important Note:
Before using the terminal emulator, you must register your Pismo Toolset. Until you do so, usage will be restricted to a 20-day trial period for the terminal emulator and other applets contained in the Toolset. To register, fill out the contents of the Registration Form, then click on the Register Now button. This will print a Registration report which, must be faxed to Innovative Integration. Innovative Integration will E-mail you an Access Code, which must be typed into the Registration Form for all the features to be enabled.

Terminal Emulator Menu Commands
The terminal emulator provides several menus of commands for controlling and customizing its functionality. These functions are available on the menu bar, located at the top of the the terminal emulator main window. Speed button

equivalents for each of the menu options are also available on the button bar located immediately beneath the menu bar. The following is a description of each menu entry available in the terminal emulator, and its effects.



The File Menu

File | Load - provides for COFF (Common Object File Format) program downloads from within the terminal emulator. When selected, a file requester dialog box is opened and the full pathname to the COFF filename to be downloaded is selected by the user. Clicking "Open" in the file requester once a filename has been selected will cause the requester to close and the file to be downloaded to the target and executed. Clicking "Cancel" will abort the file selection and close the requester with no download taking place.

This operation can optionally be initiated via the button.

File | Reload - Reloads and executes the COFF file last downloaded to the target. It provides a fast means to re-execute the application program most recently loaded into the target board.

This operation can optionally be initiated via the button.

NOTE: File | Load and File | Reload functions use the JTAG debugger and Code Composer Studio in order to effect the program download.

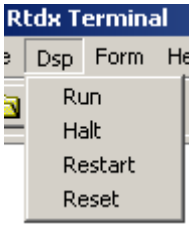
File | Save – saves the textual contents of the Terminal and Log tabs to a user specified file.

File | Print - prints the textual contents of the Terminal and Log tabs to a user specified printer.

File | Exit – closes the emulator application, terminating console emulation.

The DSP Menu

Dsp | Run - causes the terminal emulator to bring the target board into a cold-start, uninitialized condition. This is functionally identical to performing Debug | Run within Code Composer Studio.



This operation can optionally be initiated via the button.

Dsp | Halt - causes the terminal emulator to suspend DSP program execution. This is functionally identical to performing Debug | Halt within Code Composer Studio.

This operation can optionally be initiated via the button.

Dsp | Restart - rewinds the DSP program counter to the application entry point, usually *c_int00()*. This is functionally identical to performing Debug | Restart within Code Composer Studio.

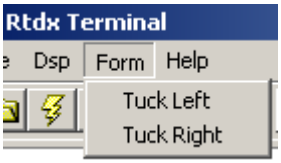
This operation can optionally be initiated via the button.

Dsp | Reset - causes the terminal emulator to bring the target board into a cold-start, uninitialized condition. This is functionally identical to performing Debug | Reset Dsp within Code Composer Studio.

This operation can optionally be initiated via the button.

The Form Menu

Form | Tuck Left - repositions the main application window to the bottom left of the Windows desktop.



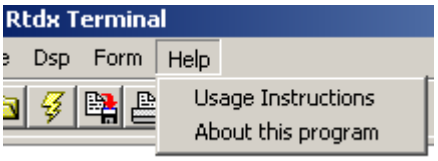
This operation can optionally be initiated via the button.

Form | Tuck Right - repositions the main application window to the bottom right of the Windows desktop.

This operation can optionally be initiated via the button.

The Help Menu

Help | Usage Instructions - displays online help detailing use of the application, including command-line arguments.



This operation can optionally be initiated via the button.

Help | About this Program - displays a dialog containing program revision and tech support contact information.

Options Tab:

The Options tab (seen below) contains controls to allow user-customization of the appearance and operation of the terminal emulator.

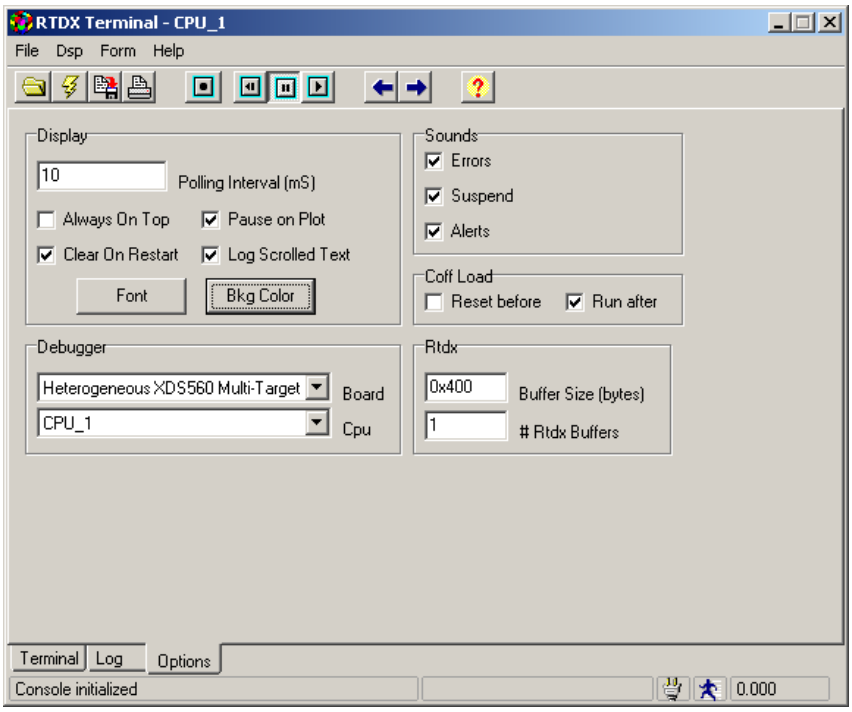


Figure 7. RtdxTerminal Options

Display *Group*
Controls within the Display group box govern the visual appearance of the terminal emulator, as detailed below.

Polling Interval - specifies the period, in milliseconds, between queries for data received from the DSP via the JTAG RTDX interface. Lower numbers increase performance but increase Host CPU load.

Always on Top - specifies that the terminal application should always remain visible, atop other applications on the Windows desktop. This check box controls whether the terminal emulator is forced to remain a foreground application, even when it loses keyboard focus. This is useful when running stdio-based code from within the Code Composer environment, when it's preferable to make terminal visible at all times. The terminal will remain atop other windows when this entry is checked. Select the entry again to uncheck and allow the terminal emulator window to be obscured by other windows.

Clear on Restart - specifies whether the terminal display and log will be automatically cleared whenever the DSP is restarted.

Pause on Plot - specifies whether standard I/O will be suspended following display of graphical information in the Binview applet which is automatically invoked via use of the Pismo library

Plot() command. If enabled, standard I/O may be resumed by clicking the button.

Log Scrolled Text - specifies whether text information which scrolls off screen on the Terminal tab is appended to the Log display. If enabled, standard I/O performance will degrade slightly during lengthy text outputs.

Font - button invokes a font-selection dialog which allows selection of user-specified font within the Terminal and Log text controls.

Bkg Color - button invokes a color-selection dialog which allows selection of user-specified background color within the Terminal and Log text controls.

Sounds Group
Controls within the Sounds group box govern the audible prompts generated by the terminal emulator, as detailed below.

Errors - if enabled, file I/O and other errors encountered during operation generate an audible tone.

Suspend - if enabled, suspension of standard I/O, such as following plotting via Binview, generate an audible tone.

Alerts - if enabled, alert conditions encountered during standard I/O, such as upon display of the ASCII bell character, generate an audible tone.

Coff Load Group
Controls within the Coff Load group box govern behavior surrounding a COFF executable download.

Reset Before - if enabled, the Code Composer Debug | Reset DSP behavior is executed before attempting to download the user-specified COFF file.

Run After - if enabled, the Code Composer Debug | Run behavior is executed immediately following the download of a user-specified COFF file.

Debugger Group
Controls within the Debugger group box specify the target DSP with which RTDX communications is established.

Board - specifies the board hosting the target DSP to be used in RtdxTerminal stdio communications. This combo box is populated with all available board types configured using the Code Composer Setup utility.

Cpu - specifies the identifier of the specific DSP to be used in RtdxTerminal stdio communications. This combo box is populated with all available CPUs present on the baseboard as configured using the Code Composer Setup utility.

Terminal Emulator Command Line Switches
The terminal emulator also provides the following command line switches to further modify program behavior. The switches must be supplied via the command line or within Windows shortcut properties (see the Installation section for more information), and will override the default behavior of the applet.

Multiple instances of the terminal emulator may be invoked simultaneously in order to support installations utilizing multiple target boards. Instances of the terminal emulator, after the first loaded instance must be configured via command line switches in order to properly communicate with their associated target.

-board *boardtype* - Use the -board switch to force an instance of the terminal emulator to communicate with a specific type of target board, boardtype. Supported board types are those configured using the Code Composer Setup utility, such as "C64xx Rev 1.1 XDS560 Emulator".

-cpu *cputype* - Use the -cpu switch to force an instance of the terminal emulator to communicate with a specific CPU on a target board. Supported CPU types are those configured using the Code Composer Setup utility, such as “CPU_1” or “CPU_A”.

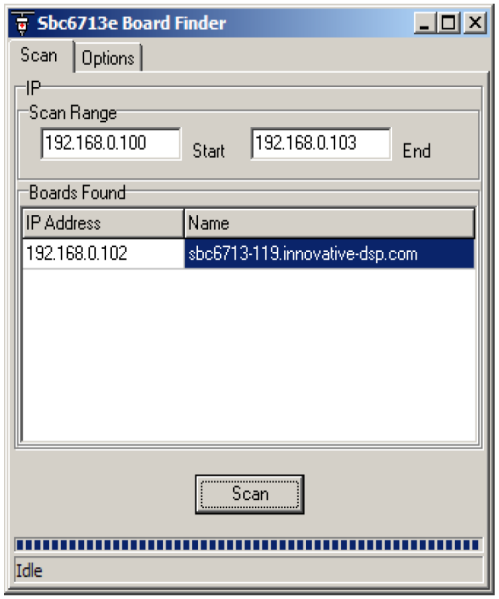
-f *filespec* - Use the -f switch to force the terminal emulator to load and run the specified COFF file. The “*filespec*” field should be a standard Windows file specification, including both the path and file name as a unit, to allow the user to force the terminal emulator to download the specified file to the target DSP board, as soon as the terminal emulator is loaded. This field is particularly useful in situations where the the terminal emulator is “shelled to” from within an other Host applications to facilitate the automatic execution of target applications employing standard I/O.

Applets for SBC6713e Baseboard

Board Finder Utility (Sbc6713eFinder.exe)

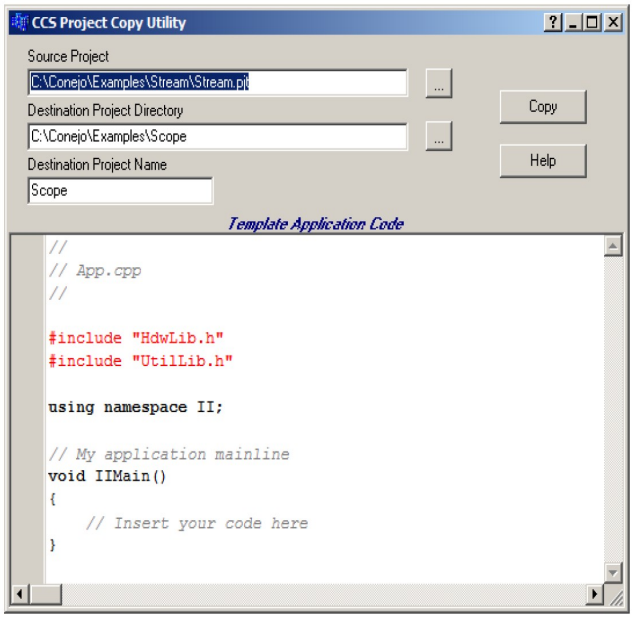
In the factory default configuration, the Innovative Integration Sbc6713e DSP board obtains an IP address from the DHCP server on your network. Consequently, it is necessary to scan the network to discover the IP address of each newly-added board. This utility serves that purpose.

Enter the Start and End addresses for the internal network to which the SBC6713e is attached. Insure that the SBC6713e is powered, and that the network activity indicator on the board is flashing, indicating network activity. Then, click the Scan button to snoop the network. All boards discovered during the scan will be listed in the Scan | Found Boards list.



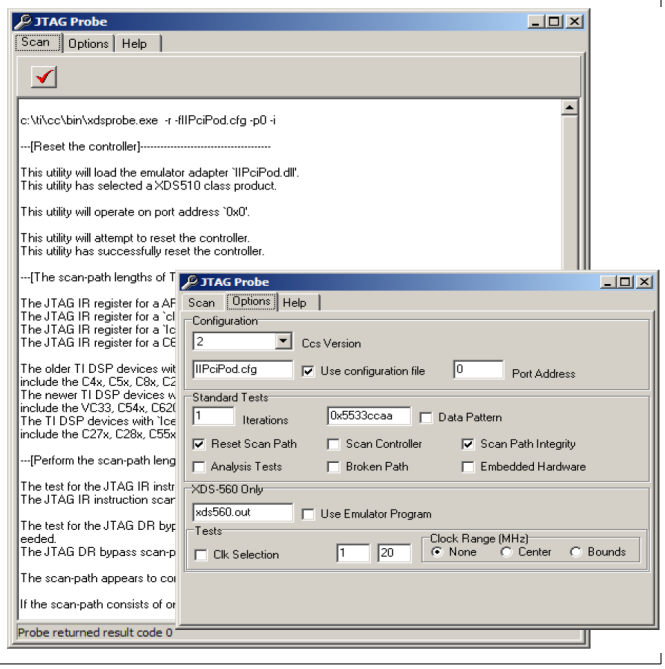
Target Project Copy Utility (CopyCcsProject.exe)

The CopyCcsProject.exe applet is used to copy all project settings from a known-good template project into a new DSP Code Composer project. This simplifies new project development, by eliminating the multi-step process of copying the myriad individual project settings from a source project in a newly-created project.



Scan Path Diagnostic Utility (JtagScanpath.exe).

The JtagScanpath.exe applet is a simple GUI front-end to the powerful Texas Instruments XdsProbe.exe command-line utility. The utility is of value in debugging JTAG debugger installation and reliability problems. The tool is capable of performing comprehensive scan integrity tests for both the Innovative Code Hammer and the TI XDS560 emulators. The complete reference to available tests and features is listed on the application Help tab.



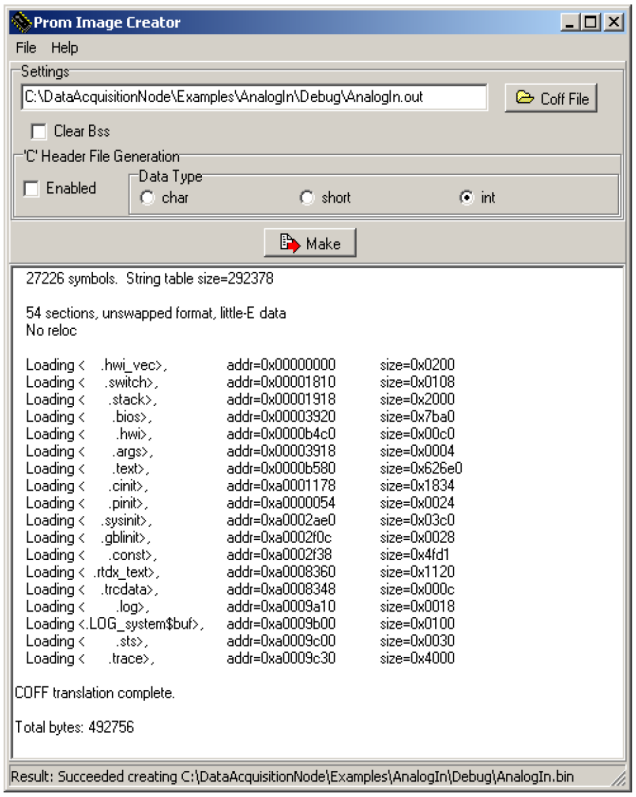
Flash Conversion Utility (PromImage.exe)

Innovative boards capable of booting from Flash ROM include a small application program called Loader which begins execution on system cold- start. If a valid Innovative Loader COFF image is identified in Flash ROM, and if the boot jumper on the board is installed, then this Loader program resurrects the COFF image into DSP memory and launches it.

The PromImage.exe applet supports conversion of COFF executable images into proprietary Innovative Loader format. This data format has been developed to circumvent a boot-limitations in the TI C6000 family, which is incapable of booting COFF images directly.

This applet automatically accommodates the endianness of the COFF image. Optionally, the .bss section of the generated image may be cleared. This increases the image size, but is generally required in order to mimic the load-time behavior of Code Composer Studio, in which the application was likely debugged.

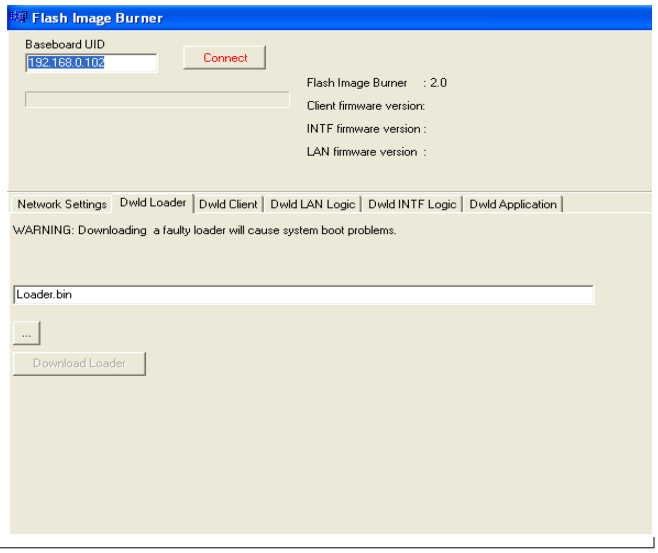
In addition to binary images, the applet also supports generation of images translated into 'C'-compatible source code containing initialized data arrays. This is useful in applications which dynamically reconfigure logic from within target application code.

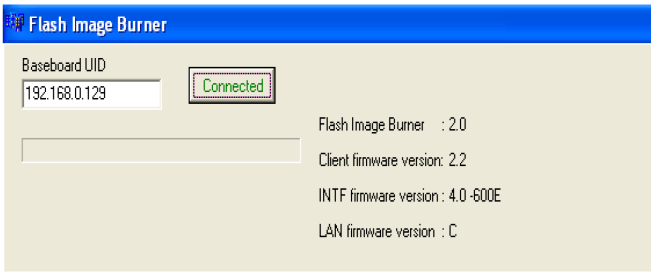


FlashBurn

FlashBurn.exe is a windows application that enables the user to update firmware (.exo logic images) or an embedded application (COFF images) into the Sbc6713e flash. It enables the user to choose whether to boot up the C6713 DSP using the user's embedded application or to boot the default Innovative Integration "benign" program. The user can reverse option at any time. This is an important feature useful when testing or deploying an embedded application.

The tool also enables use of static IP or DHCP (server-assigned) addressing of the board.



Usage FlashBurn.exe is composed of a main window and six "tab windows". The main window is located at the upper half of the application's GUI. Its purposes are: • To establish a communication with the Sbc6713e card. • Display firmware versions. • Display progress bar when downloading a program. • Display error messages.	

Connect button

Use the Connect button to establish communications with your target board. In order to establish communications, you must first enter the IP address of the Sbc6713e with which communications is to be established into the Baseboard UID combo box. Use the Sbc6713eFinder applet to discover the IP address, if necessary.

If you have an earlier version of the coprocessor (DM642) client program (i.e. before version 2.2) , the connect button will remain red, and after six seconds all download buttons (tab windows) will be enabled.

This happens because earlier client programs did not support two communication sockets added starting with version 2.2. Despite this behavior, this tool is still useful when used in conjunction with the earlier client versions. Once you update your board to v2.2 of the client program, FlashBurn.exe will be able to display the revision of the firmware components in the Sbc6713e card.

Below is a screen shot of how it looks when connected to a Sbc6713e with Client version 2.2.

Network Settings tab

On Network Settings tab you may choose to use static IP settings or use DHCP protocol where your IP sever will assign an IP address to the SBC6713e card.

To enable use of a static IP address, enable the Static IP checkbox, fill in the static IP information located in the Settings group box, then press the Update IP settings to ROM button. You may switch between the two at any time. Once you update, close FlashBurn.exe application and re-open to establish communications with the card again (i.e. it has new IP address).

If your board has Client program 2.2 or later and you have run the RtdxTerminal program "FlashLoad.out" on the DM642 DSP (the network co-processor) you may follow the following procedure to find your board's IP address:On a

DOS prompt window simply ping:

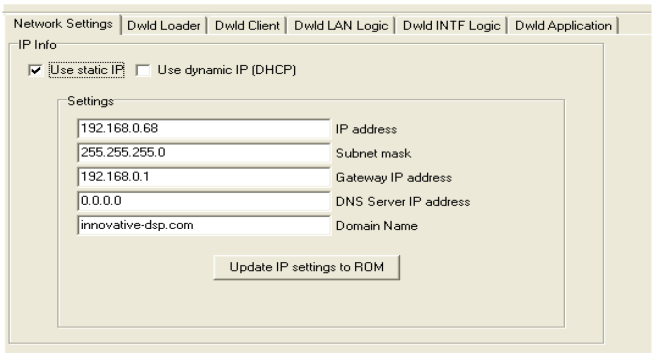
sbc6713-{Sbc6713e PCB rev}{Serial Number}-innovative-dsp.com

So if the PCB rev is "C" and Serial Number is 105, then ping:

Sbc6713-C105.innovative-dsp.com

The DOS prompt will display the IP address of the card between square brackets.

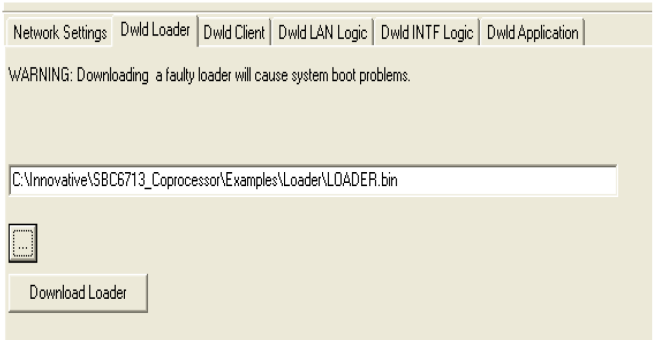
Below is a screen shot of Network Settings tab.



Download Loader tab

This is a tab that is not usually used, unless instructed by Innovative Integration's release notes or Tech Support. This tab will update the loader program which launches the DM642 client application during cold-start initialization. If an inappropriate binary file is loaded using this tab window, your card will have to be returned for repair. Burning an incorrect Loader program will cause the card not to boot up and not to be accessible via Code Composer®.

Please read carefully the label of each button you may press, all of them are labeled differently to avoid this type of errors. Below is a screen shot of this tab.

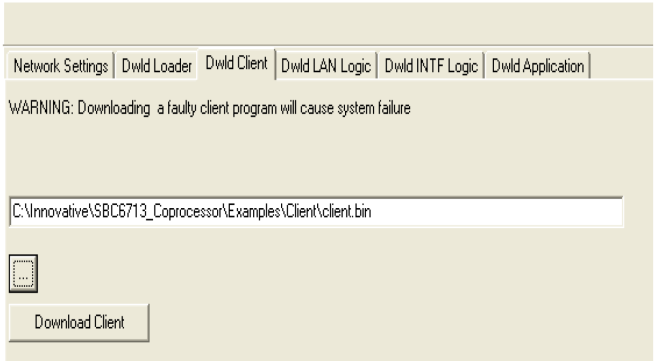


Download Client tab

This is another infrequently-used maintenance function to be employed only when instructed by the release notes or by Innovative Integration's Tech Support.

This tab updates the DM642 coprocessor client program. This program is in charge of all Host <-> Target communications, loading of firmware into the two FPGAs at startup and launching of C6713 applications. The format of the client program is a binary image of a COFF file, produced using the Innovative PromImage applet.

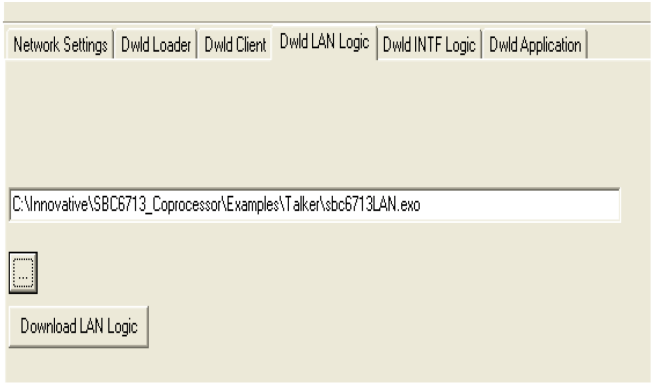
Below is a screen shot of Download Client tab.



Download LAN logic firmware

This tab is used to upgrade the LAN logic device's firmware. This logic image should be updated only under specific instructions by Innovative Tech Support using an Innovative-supplied firmware image. If this firmware image is corrupted, the consequences are not as bad as with the Client and Loader programs, since the LAN firmware can be reprogrammed using CCS running ReLANlogic.out. If you are working on a Rev C PCB, then CCS® will need to be setup again to by-pass the C6713 DSP with a scan depth of 46, the C6713 is the first DSP on the scan path. Also before booting the SBC6713e card, a jumper needs to be in place (JP16).

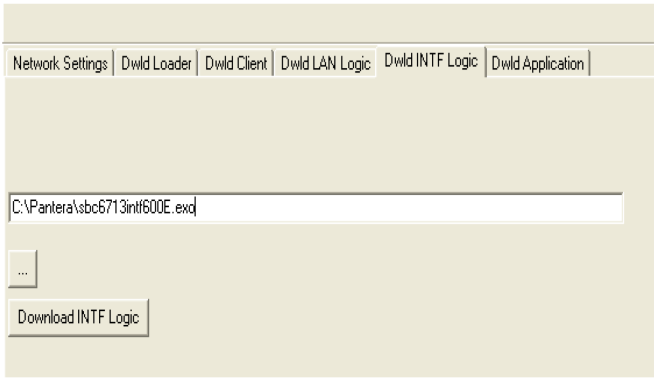
Below is a screen shot of Download LAN firmware tab window.



Download INTF logic firmware

This tab may be routinely changed by users planning on expanding/modifying on the the Sbc6713e logic. If for some reason the exo file you downloaded makes the card inoperable, just download again a known good exo file or if the card is unable to boot, follow the procedure described above, where you would have to run the ReLogic_600.out or ReLogic_300.out (depending on the board configuration) from CCS® (Code Composer Studio ®)

Below is a screen shot of Download INTF firmware tab window.



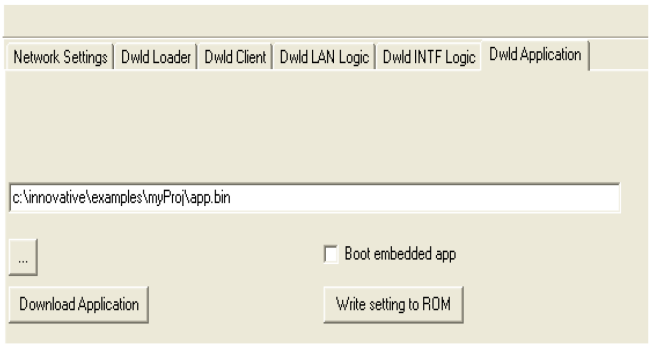
Download Application

This window is capable of loading an application program that will run on the C6713 DSP once it boots. The board may be configured to either boot your embedded application or the default Innovative Integration “benign” program via the Boot embedded app checkbox.

To enable use of a user-supplied embedded application, use PromImage to convert your application to binary bootable format. Then browse to this image using the ... button. Burn this image into flash using the Download Application button. Finally, enable the Boot embedded app checkbox and click Write setting to ROM.

If for some reason your application does not behave the way you expected, you can always revert back to use the benign program by simply unselecting the check box and then by pressing the Write setting to ROM button.

Below is a screen shot of this tab window.



Custom Logic Development

Embedded data acquisitions, servo control, stimulus response and signal processing jobs are easily solved with the SBC6713e baseboard using the supplied Pismo software. There are wide selection of peripheral devices available as plug in omnibus modules, for many types of signals from DC to RF frequency applications or audio processing.

SBC6713e is a high performance, flexible, stand-alone DSP board with Ethernet connectivity, loaded with I/O peripherals, 300Mhz floating point C6713 DSP gives an open platform with several omnibus available off the shelf daughter cards with a wide choice of A/D and D/A and also supporting simple EMIF bus interface to custom I/O daughter cards. System-level integration is facilitated with on-board digital, DDS timebase, external clock input, multi-card sync, FPDP port data links, 4MB flash ROM and watchdog. One Spartan-IIE FPGA of up to 600Kgates is available for implementing custom logic interfaces and accelerated signal processing. In the FPGA it is possible to process data at very high rates.

Begin by analyzing your application to identify the operations that are high speed and lower speed. Higher speed signal processing operations should be targeted at the FPGA provided that they are of manageable complexity. Typical FPGA operations include FIR filters, down conversions, specialized high speed triggering and data sampling and FFTs. All of these functions are deterministic mathematical functions that are suitable for the FPGA. Data formatting protocols and control functions are typically more easily implemented on the DSP.

The SBC6713e has two FPGAs on it: Xilinx Spartan IIE (50K gates) and Xilinx Spartan IIE (600K system gates). The Spartan IIe (600K gates) is used for the analog interfacing, and as the computational logic on the board. Whereas Spartan IIe(50K gates) behaves as an Inter-processor FIFO between TMS32DM642 and TMS320C6713 and full interrupt support for DMA transfers from either processor, the logic for this Spartan IIe is not provided.

The analog logic which resides in Spartan IIE (300K-600K gates) is provided in source form to assist in custom application developments. The FPGA logic images may be downloaded to Spartan IIe as part of the software application using fast Ethernet.

Overview

The SBC6713e Logic Frame work is the basis for custom logic application development. This Framework allows the FPGA designers to begin with a known working design that illustrates analog IO and DSP interfacing techniques. The custom application is then fit within the framework and includes the application specific code. In many cases, the standard methods for moving data to the DSP are retained so that custom software is minimized.

Target Devices

SBC6713e has as its user programmable logic a Xilinx Spartan IIE (600K gates) FPGA. The Spartan IIe FPGA have many resources in addition to the logic gates such as embedded RAM blocks, delay- locked loop (DLLs) and flexible IO standards that are used in the provided logic. The Xilinx device used for analog logic is XC2S600E-6FG456C.

Code Development Tools

Developing logic code for this FPGA is done using the Xilinx ISE Toolset and the Framework logic provided. The Framework logic is delivered in VHDL language, along with the control files for the Xilinx ISE software to allow the user to modify and compile the logic.

Simulation files are provided for use with MentorGraphics ModelSim. A macro for the testbench is provided to compile and load the simulation. When using ModelSim, it is required to compile and reference the library files for unisim, simprim, and xilincorelib for the simulation to load properly.

Steps to Successful Use

1. You must have a simulation tool. Innovative provides files for ModelSim. If you don't simulate, it is unlikely you will successfully complete the logic design within your lifetime.
2. Estimate the logic usage for your application.
3. Define the clock domains in the design and methods to transition them. FIFOs work great.
4. Identify crucial timing constraints. Plan the logic method to meet them. Pipelining is usually a good approach.
5. Until your simulation works, your hardware will not work. Debugging on real hardware is very timing consuming. Perform real timing simulations wherever possible.

Using FPGA Framework

As a starting point, here is a big overview of what is attached to the Spartan IIe FPGA.

Figure 8. FPGA Framework Overview

The Framework Logic block diagram is shown below.

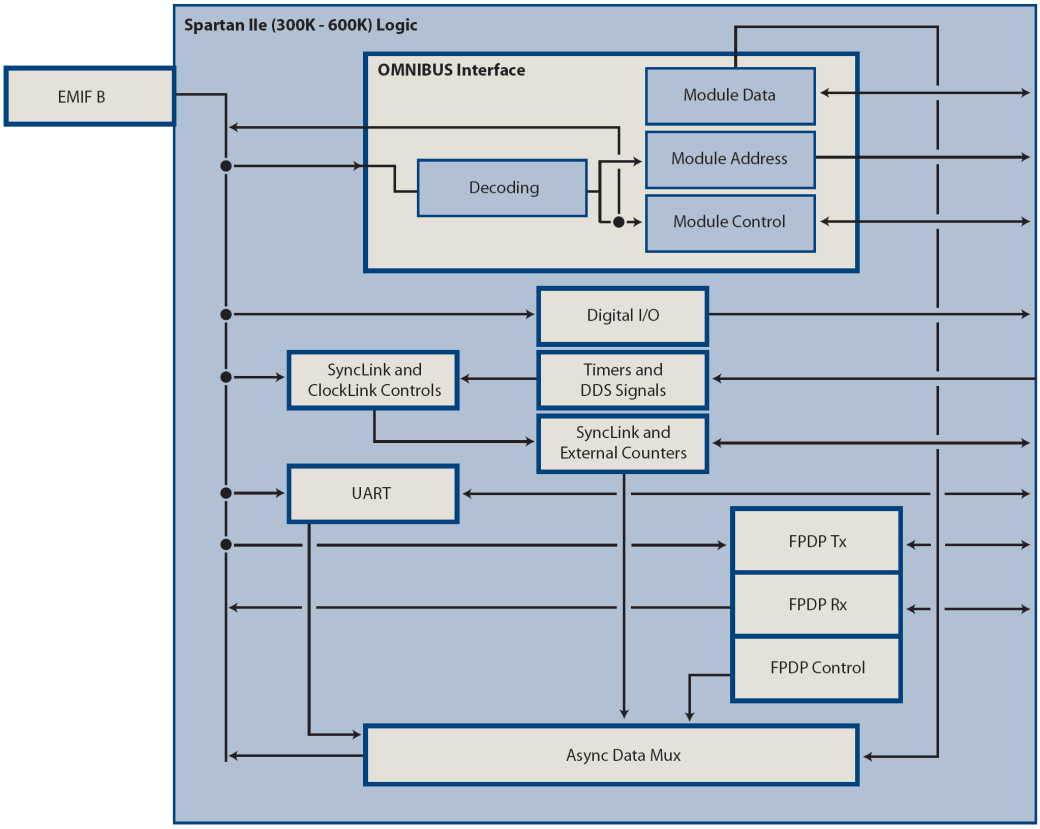
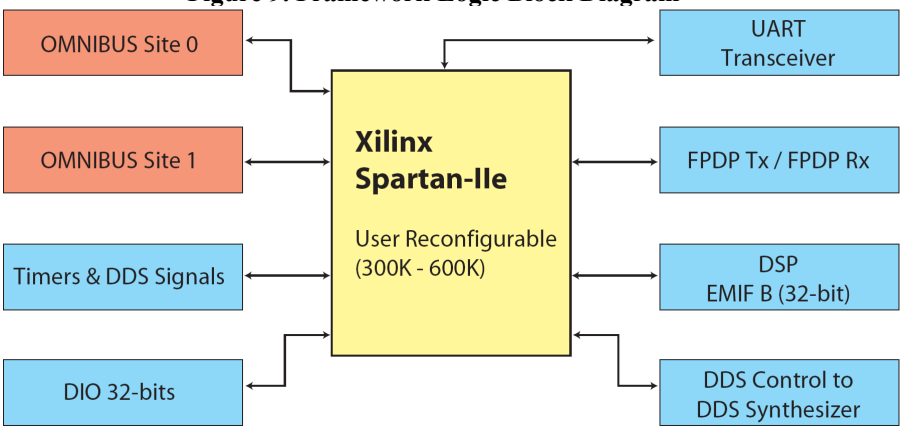


Figure 9. Framework Logic Block Diagram



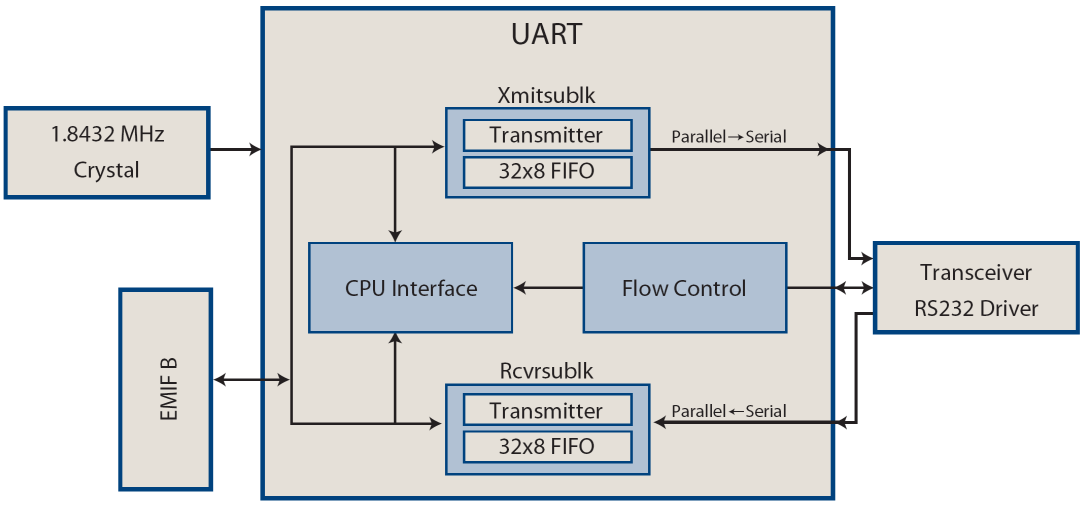
As can be seen in the block diagram, the EMIF B is used as the interface to the analog I/O. The EMIF B memory space is decoded so that control and configuration registers are mapped as well as burst FIFOs for the high speed data paths. Please refer to memory map (Table 13) of SBC6713e for external peripherals and memory (see TI peripheral manual for TMS320C6713 for on chip peripherals) for the Framework Logic. Most designers integrate application-specific features into the standard memory mapping to preserve as much software as possible. Also, the memory decoding and data interface to the DSP are architected in the logic to support the burst or asynchronous memory interfaces.

Note: SBC6713e External Memory Map (Table 13).

UART Overview (RS232 Asynchronous Serial Port)

The SBC6713e implements a single asynchronous full duplex serial port channel with RS232 drivers compatible with standard PC serial port interfaces. The serial port is implemented in the application FPGA as a logic function and uses a RS232 transceiver.

Figure 10. UART Block Diagram.



Developing FPGA Firmware

Spartan IIe is designed for the SBC6713e using VHDL coding and were compiled under Xilinx ISE 6.2.3, XST synthesis. The example/debug logic design is a project loadable by opening ISE and preforming a Project | Open on the sbc6713_intf.npl file in the logic root directory. The design directory path originally used to house this project hierarchy was c:\projects\sbc6713\intf_logic. It is recommended that the customer's installation duplicate this directory configuration in order to avoid problems with Xilinx tools.

Set up for the environment is included in the npl file. Constraints for the example logic are sbc6713_top.ucf and should be applied to the sbc6713_intf.vhd file, which is the top of the design. The constraints have the pin assignments, I/O standards and some basic timing constraints for the framework. Compilation is usually easy and users who are familiar with Xilinx ISE can start by verifying that the provided design recompiles. Just click on the synthesis, then Implement then generate program

ming file buttons to get through the design process. Watch out for the warnings, such as timing constraints violation, as these may really mean that your design would not work at speed. These are viewed in the report files. The tool issues many warnings along the way, these can be ignored for the sample logic.

Note: If you are unfamiliar with Xilinx ISE, see the Xilinx tutorial on using it, accessed from the help button on the top of the application.

Once you have successfully complied and installed the software, your development can begin by editing out code you don't need and deleting the associated constraints. You will have to keep the constraints associated with pin all assignments whether or not you use that function, as pins are fixed by the board design.

Here is a summary of the steps to follow:

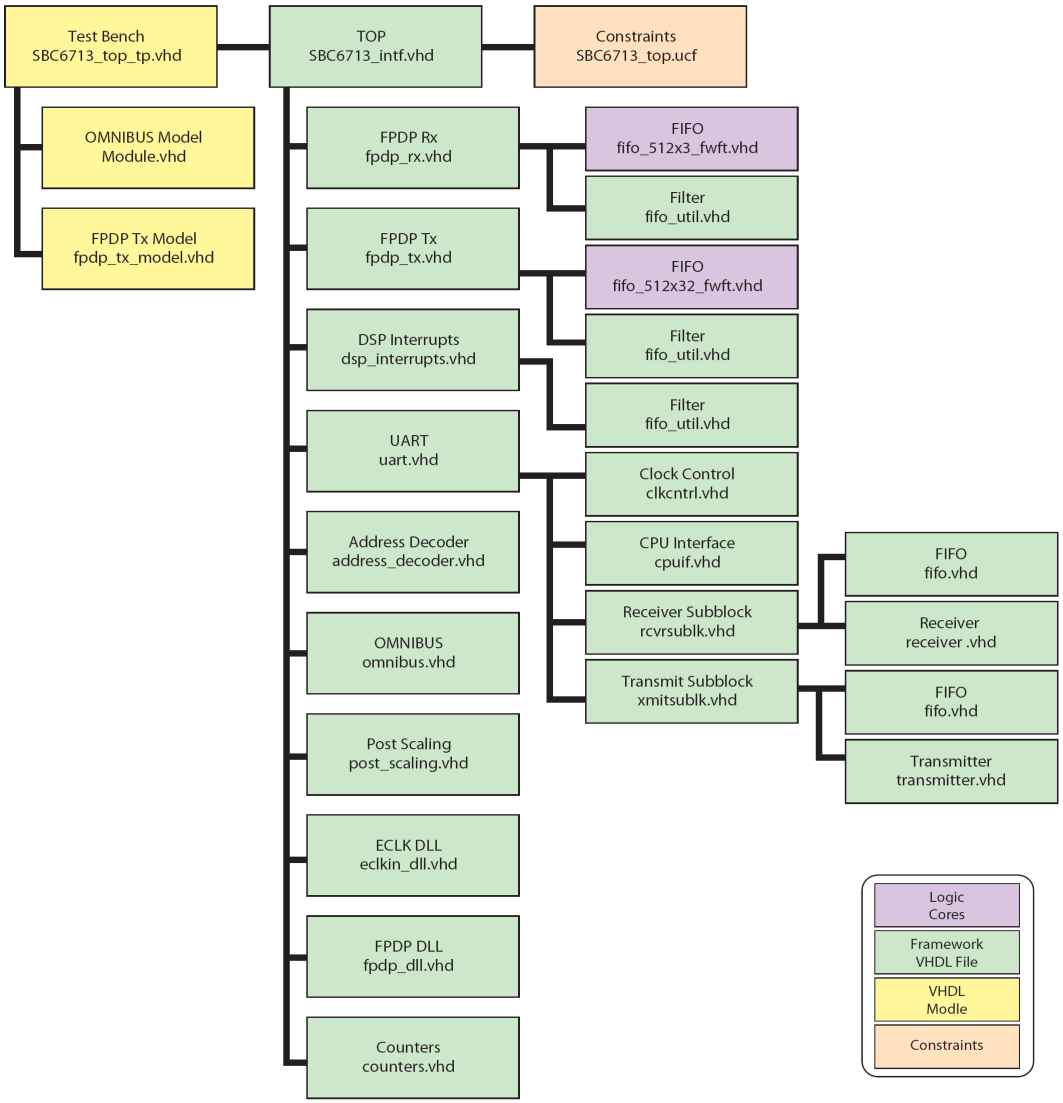
Steps to follow to logic Builds

1. Install Xilinx ISE 6.0 or above.
2. Unzip the Framework Logic into a suitable directory.
3. Open the ISE project sbc6713_intf.npl
4. Recompile and refit the logic. The resultant output file will be sbc6713_intf.bit. Intermediate report files are sbc6713_intf.bld (NGD Build report), sbc6713_intf.mrp (MAP report), and sbc6713_intf.par (Place and Route Report). The .PAR file shows a timing summary. None of these files should include errors, only warnings. The warnings can be safely ignored.
5. Generate a PROM image as an .exo file using Xilinx Impact. Use one bitstream, starting at 0x0, with Motorola S-Record (EXO) format for the bit file sbc6713_intf.bit. The PROM size my be set to auto.

Logic Files and Hierarchy

Within the ISE environment, the logic for the SBC6713e has a structure as shown in the figure below.

Figure 11. Logic Files and Hierarchy



Fitting Results for the Framework Logic

The Framework logic consumes about 21% of the logic for the Spartan IIe (600K) device. The results of the mapping process, as taken from the sbc6713_top.mrp report, are as shown. Notice the memory consumed during the compile is 123MB for this design.

```
Design Summary
--------------
Number of errors:      0
Number of warnings:   13
Logic Utilization:
  Total Number Slice Registers:   2,942 out of 13,824   21%
    Number used as Flip Flops:                2,939
    Number used as Latches:                       3
  Number of 4 input LUTs:         3,166 out of 13,824   22%
Logic Distribution:
    Number of occupied Slices:                       2,655 out of  6,912   38%
    Number of Slices containing only related logic:  2,655 out of  2,655  100%
    Number of Slices containing unrelated logic:         0 out of  2,655    0%
        *See NOTES below for an explanation of the effects of unrelated logic
Total Number 4 input LUTs:        3,435 out of 13,824   24%
      Number used as logic:                   3,166
      Number used as a route-thru:              171
      Number used as Shift registers:            98
   Number of bonded IOBs:           312 out of    325   96%
      IOB Flip Flops:                           170
      IOB Latches:                               18
   Number of Block RAMs:             12 out of     72   16%
   Number of GCLKs:                   4 out of      4  100%
   Number of GCLKIOBs:                2 out of      4   50%
   Number of DLLs:                    2 out of      4   50%
   Number of BSCANs:                  1 out of      1  100%

   Number of RPM macros:            2
Total equivalent gate count for design:  271,664
Additional JTAG gate count for IOBs:  15,072
Peak Memory Usage:  123 MB
```

Figure 12. Map Results

The Place and Route Step results are taken from the sbc6713_top.par report. Timing analysis is shown for the design. It is important to review this report to find timing errors. The PAR report shows how the design performed against each defined timing constraint. For further analysis, the timing analyzer tool can be used to pinpoint the source of each problem.

```
The NUMBER OF SIGNALS NOT COMPLETELY ROUTED for this design is: 0

   The AVERAGE CONNECTION DELAY for this design is:        1.932
   The MAXIMUM PIN DELAY IS:                             10.608
   The AVERAGE CONNECTION DELAY on the 10 WORST NETS is:   8.674

   Listing Pin Delays by value: (nsec)

    d < 2.00   < d < 4.00  < d < 6.00  < d < 8.00  < d < 11.00  d >= 11.00
   ---------   ---------   ---------   ---------   ---------   ---------
       11650        5332        1516         369          41           0

Timing Score: 441574

WARNING:Par:62 - Timing constraints have not been met.

Asterisk (*) preceding a constraint indicates it was not met.
   This may be due to a setup or hold violation.

--------------------------------------------------------------------------------
```

```
  Constraint                                | Requested  | Actual     | Logic
                                            |            |            | Levels
--------------------------------------------------------------------------------
  TS_J_TO_J = MAXDELAY FROM TIMEGRP "J_CLK" | 30.000ns   | 14.022ns   | 5
   TO TIMEGRP "J_CLK" 30 nS                 |            |            |
--------------------------------------------------------------------------------
  TS_U_TO_J = MAXDELAY FROM TIMEGRP "U_CLK" | 15.000ns   | 5.062ns    | 1
   TO TIMEGRP "J_CLK" 15 nS                 |            |            |
--------------------------------------------------------------------------------
* TS_FPDP_Rx_Control = MAXDELAY FROM TIMEGR | 10.000ns   | 12.055ns   | 7
  P "FPDP_Rx_Control" TO TIMEGRP "FFS"  10  |            |            |
  nS                                        |            |            |
--------------------------------------------------------------------------------
```

Figure 13. PAR Results

Adding functionality to the Framework Logic

The framework logic is a starting point for the more advanced logic that will be your SBC6713e application logic. It is suggested that you begin by simply recompiling this logic and verifying that you can recreate the framework logic as delivered (see steps above). This will verify that you have all the libraries and FPGA compilation tools required to move ahead on your design.

Once you have successfully recompiled the logic, it is now possible to begin adding and replacing the simple logic with your application code. This is done by modifying the top VHDL to include your sub- functions, then modifying the test bench code to adequately stimulate your design.

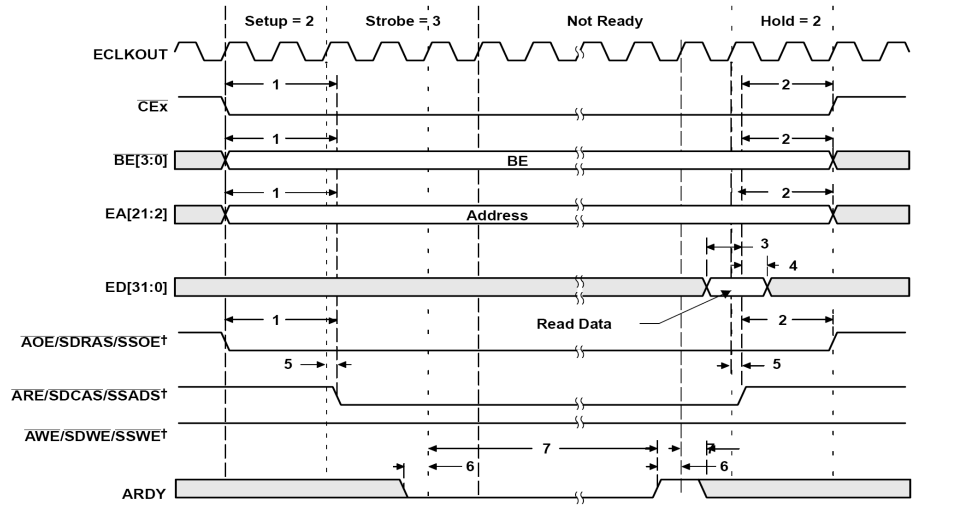
Innovative Integration strongly recommends that you fully simulate your design before putting this logic into the SBC6713e (Spartan IIe 300K or 600K) FPGA. This will not only save time in debugging, but could also prevent simple errors from causing serious damage to the module. A tool like ModelSim is generally required for this high density, complex logic design that give full visibility into the logic behavior prior to actual synthesis.

Many pre-written logic functions are available to assist in logic development from Xilinx and other vendors. These logic functions may be viewed at the Xilinx web site (http://www.xilinx.com/ ipcenter/index.htm). These logic functions include basic math, filters, FFTs and other functions that are useful in logic designing with the Xilinx Spartan IIe FPGAs.

Logic Design Methods and Code Development

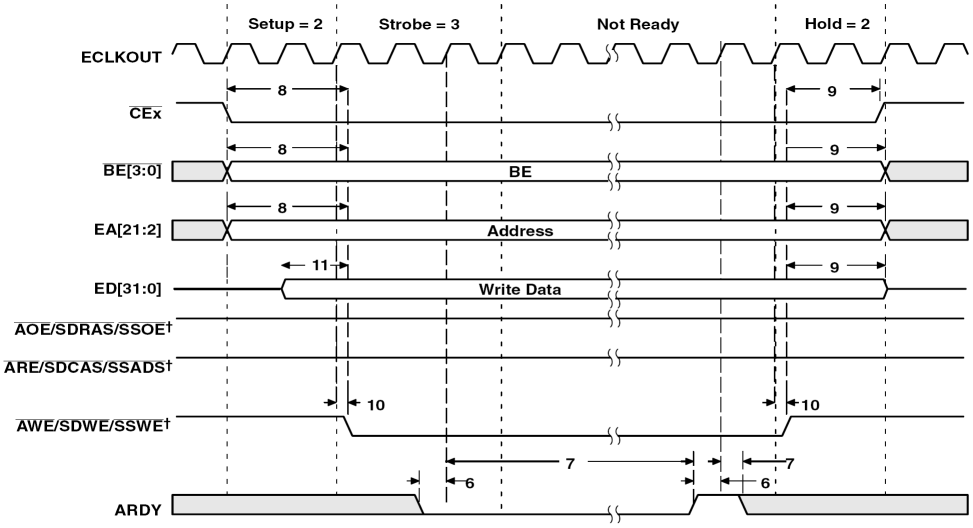
DSP Interface Logic

The SBC6713e EMIF B memory decodings are arranged so that CE0 is Synchronous DRAM (SDRAM), CE2 is a burst memory space for FPDP FIFO and the Inter-processor FIFO for the ‘6713 is mapped to CE2 as a burst memory device allowing very high rate transfers with the DM642, whereas CE1 and CE3 are Asynchronous and the wait states are controlled by hardware ready. Generally speaking, the async peripherals are not used as high speed devices since this is inherently a slow access protocol - the sync burst memory interface is at least 4x faster. The following diagrams show the DSP access timing for burst reads and writes.



† AOE/SDRAS/SSOE, ARE/SDCAS/SSADS, and AWE/SDWE/SSWE operate as AOE (identified under select signals), ARE, and AWE, respectively, during asynchronous memory accesses.

Figure 14. DSP Asynchronous Read Timing (Courtesy of Texas Instrument)



† AOE/SDRAS/SSOE, ARE/SDCAS/SSADS, and AWE/SDWE/SSWE operate as AOE (identified under select signals), ARE, and AWE, respectively, during asynchronous memory accesses.

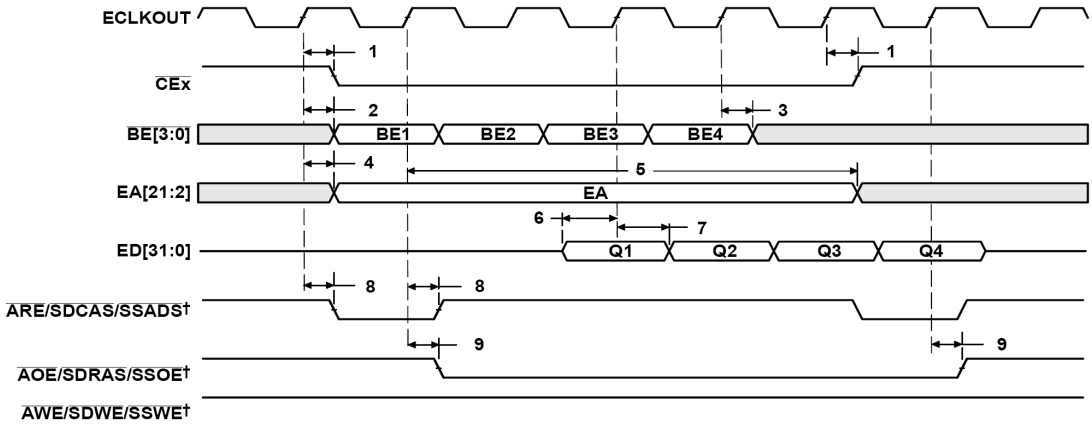
Figure 15. DSP Asynchronous Write Timing (Courtesy of Texas Instrument)

For slow speed devices such as configuration and control registers, asynchronous access are used in the Framework Logic. Asynchronous accesses provide the most flexibility in timing control and are the easiest to use in most designs, albeit the slowest. The EMIF control registers in the DSP allow the programmer to define SETUP, STROBE and HOLD lengths for the

cycle that give a programmability to the access timing. For more control, the ARDY signal allows the logic designer to insert additional wait states to the STROBE timing as needed.

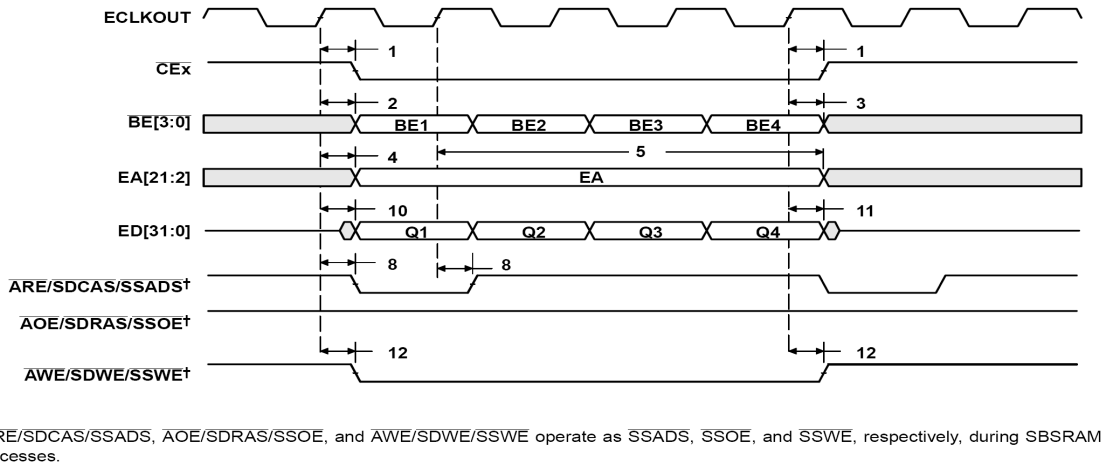
For the high speed data paths in the Framework logic, burst accesses from the DSP provide the highest data rates. The EMIF configuration registers are set for SBSRAM memory interface timings, and in the logic the FIFOs respond to these signals to deliver data in continuous bursts. In the burst mode, one data point (32-bits wide) is provided for each clock. As can be seen from the read and write burst timing diagrams, data is at least two cycles latent from the control signals for reads, and may be zero for writes. The Framework Logic uses a latency of X for reads and X for writes.

Data bursts can be of any length and the Framework logic accommodates any burst length needed. Normally, this is set by the DMA channel.



† ARE/SDCAS/SSADS, AOE/SDRAS/SSOE, and AWE/SDWE/SSWE operate as SSADS, SSOE, and SSWE, respectively, during SBSRAM accesses.

Figure 16. DSP SBSRAM Read Timing (Courtesy of Texas Instruments)



† ARE/SDCAS/SSADS, AOE/SDRAS/SSOE, and AWE/SDWE/SSWE operate as SSADS, SSOE, and SSWE, respectively, during SBSRAM accesses.

Figure 17. DSP SBSRAM Write Timing (Courtesy of Texas Instruments)

Since there are very few timing adjustments in the DSP EMIF control for sync registers, logic designers should be aware that burst interfaces require that the logic levels be minimized to meet timings. The SBC6713e logic has a simple data decoding and mux structure that allows the burst memories high speed, while penalizing slower async devices with a extra cycles for decoding and data delivery. Adding new read-back registers therefore should be done in a small memory region, requiring minimum logic.

The DSP memory is easily subdivided into memory types, such as async and burst by using different CE spaces. The DSP gives four CE signals for each EMIF that have timings as defined by the software (EMIF control registers), please refer to the Memory Map (Table 13).

DSP EMIF Data Read Interface

The Framework logic is designed to support full data rate from the the high speed devices, with the slower speed devices subordinate. The following diagram shows an overview of the logic for the EMIF data path.

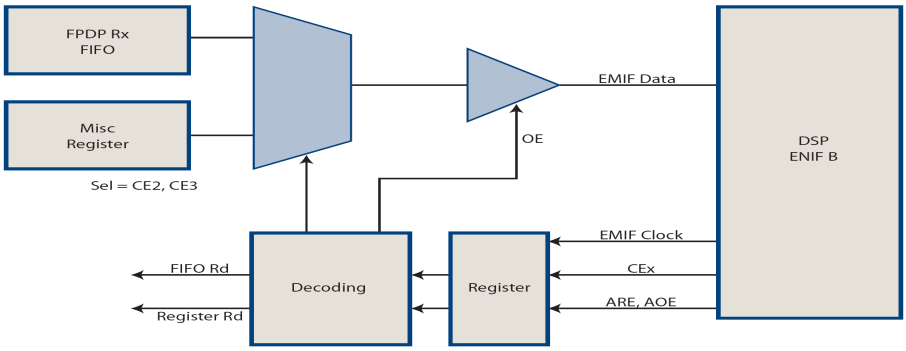


Figure 18. EMIF B Data Bus Read Interface Diagram.

The first thing to notice in the diagram is that the control signals are all immediately registered as they enter the logic. This improves the timing by grabbing the signals at the IOB registers, and allowing the internal decoding to use the registered signals. A simple decoding of the memory spaces then enables the FPDP Rx data FIFO or the ASYNC_DATA sources. The async data sources have been selected from a large case statement, using minimal address decoding, that results in a registered data source (ASYNC_DATA) that is fed to the final data mux.

In the Framework Logic code, the async data mux is as shown.

```
process(async_data_mux,status_reg,nmi_status,int4_status,dio_q,uart_int_reg,uart_d,
fpdp_tx_bitio_in,mod_din, timer2_rd, timer1_rd, timer0_rd)
begin
               case async_data_mux is
                      when "00000000000000" =>
                             async_data <= mod_din;
                      when "11000000000000"=>
                             async_data <= status_reg;
                      when "10100000000000" =>
                             async_data <= nmi_status;
                      when "10010000000000" =>
                             async_data <= int4_status;
                      when "10001000000000" =>
                             async_data <= dio_q;
                      when "10000100000000" =>
                             async_data <= uart_int_reg;
                      when "10000010000000" =>
                             async_data <= uart_d;
                      when "10000001000000" =>
                             async_data <= fpdp_tx_bitio_in;
                      when "10000000100000" =>
                             async_data <= fpdp_rx_frame_count;
                      when "10000000010000" =>
                             async_data <= (X"0000000" & "00" & fpdp_tx_pio);
                      when "10000000001000" =>
                             async_data <= (X"0000000" & "00" & fpdp_rx_pio);
                      when "10000000000001" =>
                             async_data <= (X"00" & timer_cnt0_q);
                             when "10000000000010" =>
                             async_data <= (X"00" & timer_cnt1_q);
              when "10000000000100" =>
                             async_data <= (X"00" & timer_cnt2_q);
                     when others =>
```

```
                                async_data <= mod_din;
                end case;
end process;
```

The final data mux picking between the FPDP Rx FIFO data or the asyc data is:

```
-----------------DSP Data Out----------------

process (reset, e_clk, ce2_n,ea)
begin
        if (reset = '1') then
                dsp_data_out <= (others => '0');
        elsif (rising_edge(e_clk)) then
                if (ce2_n = '0' ) then
                        dsp_data_out <= fpdp_rx_do;
                else
                        dsp_data_out <= async_data;
                end if;
        end if;
end process;
```

Notice that the data is selected simply on the basis of the registered EMIF B CE2 signal. Decoding for the read data is thus fast and simple.

As a last step, the data is enabled to drive the data bus for the reads. This is done using a simple tri-state output buffer (OBUFT)

```
ed(i) <= dsp_data_out(i) when (dsp_data_oe_n(0) = '0') else 'Z';
```

The output enable signal is a decode of the registered control signals

```
dsp_data_oe_n(i) <= '0' when (aoe_select='1') else '1';
```

Figure 19. Data Mux.

This DSP read interface has been successfully used in the Framework Logic to provide data to the DSP at burst rates using a 75 MHz EMIF clock. The pipelined nature of the code allows the FPGA to meet timing and reduces many timing issues encountered because of the complex nature of the readback mechanism. The latency for the Framework Logic is X cycles, as set in the EMIF control registers.

DSP EMIF Write Interface

In general, writes are much easier than reads because we can pipeline the control signals and data to reduce the timing problems. In the Framework logic, the incoming data is immediately registered in the IOB pins to catch the data.

```
---------Incoming DSP Data-------------
process (reset, e_clk, ce1_n_q, ce3_n)
begin
        if (reset = '1') then
                pdo <= (others => '0');
        elsif (rising_edge(e_clk)) then
                pdo <= ed;
        end if;
end process;
```

Figure 20. Incoming DSP Data.

All the control signals from DSP get registered in address decoder module. Example is shown here.

```
---------------DSP Controls-----------------------
process (e_clk, reset) begin
             if (reset = '1') then
                ce1_n_q(i) <= '1';
                awe_n_q(i) <= '1';
                are_n_q(i) <= '1';
                        aoe_n_q(i) <= '1';
             elsif (e_clk'event and e_clk = '1') then
                ce1_n_q(i) <= ce1_n;
                        awe_n_q(i) <= awe_n;
                are_n_q(i) <= are_n;
                        aoe_n_q(i) <= aoe_n;
             end if;
        end process;
end generate;
```

Figure 21. DSP Controls Signals.

The FIFOs in the logic are implemented for FPDP interface using Xilinx CoreLib functions. The FIFO component VHDL instantiation is shown here.

```
component fifo_512x32_fwft
Port (
        reset                   : in std_logic;
        din                             : IN std_logic_VECTOR(31 downto 0);
        wr_en                   : IN std_logic;
        wr_clk                  : IN std_logic;
        rd_en                   : IN std_logic;
        rd_clk                  : IN std_logic;
        fifo_reset              : IN std_logic;
        dout                            : OUT std_logic_VECTOR(31 downto 0);
        full                            : OUT std_logic;
        not_empty                       : OUT std_logic;
        wr_count                        : OUT std_logic_VECTOR(8 downto 0);
        rd_count                        : OUT std_logic_VECTOR(8 downto 0)
         );
end component;
```

Figure 22. FIFO Component Instantiation.

The FIFO interrupt filter, as contained in the fifo_util.vhd file, is shown in use here.

```
---------------FIFO Level Filtering------------------
inst_fifo_rd_int_filt: fifo_level_flt
   Port Map  (
      rst                           => fpdp_reset,
      clk                           => clk,
      fifo_en             => fpdp_fifo_rd,
      level_in          => fpdp_fifo_threshold_int_r,
      level_out          => fpdp_fifo_threshold_int,
        active_lev      => vcc,
      burst_len          => rx_fifo_burst,
```

```
new_len          => fpdp_rdfifo_control_wr,
tp                           => open
);
```

Figure 23. Using the FIFO Interrupt Filter.

This filter component takes in the threshold level interrupt as fpdp_fifo_threshold_int_r and asserts it until the a_burst length is read from the FIFO. The interrupt given to the DSP DMA controller is fpdp_fifo_threshold_int. A counter in the code counts to the burst length using the ad_fifo_rd signal.

Decoding for the FPDP Rx read FIFO accesses is a simple register that is true when CE2, and AOE are true and when DSP address (21) is not selected.

```
--------------------------DSP Burst Read-----------------------------------
dsp_ce2_burst_rd <= '1 'when (( aoe_n = '0') and (dsp_addr(21) = '0') and (ce2_n='0' )) else '0';

--------------------------DSP Burst Write----------------------------------
dsp_ce2_burst_wr <= '1' when  (ce2_n_q = '0' and awe_n_q(0) = '0' and dsp_addr_q(21) = '0') else
'0';
```

Figure 24. DSP Burst Read and Writes.

Digital IO from the Spartan IIE.

There are 32 bits of digital IO from the Spartan IIE to the IDC50 connector. These are configured as a simple LVTTL register, with software-programmed direction on a nibble basis, in the Framework logic. A number of logic standards can be programmed into the Spartan for high speed use such as LVDS and low voltage TTL as constrained by the Spartan IIE chip pins.

These pins are 5V compatible! Protection devices on each pin limit the voltage to 5V or less to prevent damage to the Spartan IIE from overvoltage or electrostatic damage. Input signals above 5V will be clamped to 5V and can sink ~1A of current for short periods. Do not use this as a signal limiting mechanism however since it is meant for protection only.

Clock Domains

Judicious choice of clock domain boundaries, and careful handling of any transition across the clock boundaries is crucial to a reliable design. Past experience has shown that more problems occur on this topic than any other. In the SBC6713e design, the EMIF clock is 75Mhz fixed rate, EMIF clock goes through DLL, which provide zero propagation delay, low clock skew between output clock signals distributed throughout the device, and advanced clock domain control. These dedicated DLLs can be used to implement several circuits which improve and simplify system level design.

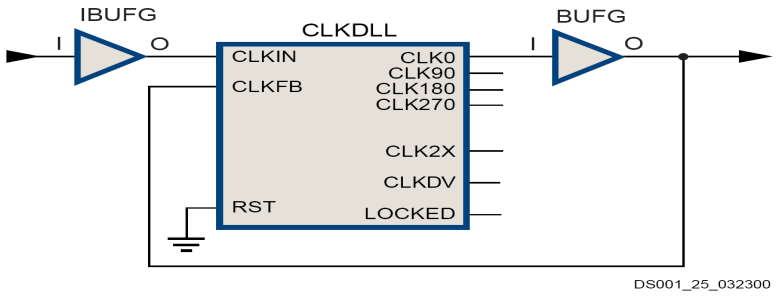


Figure 25. BUFGDLL Schematic (Courtesy of Xilinx).

Constraints

There are several important classes of constraints used by the Framework Logic : timing, pin placement and IO standards. These constraints are shown in the .ucf (user constraint file) that is used during the fitting process.

Timing Constraints

The timing constraints defined cover the clocks used in the design and the external device signal timing. Clock period period constraints are used to cover most of the logic since they define the clock rate for all flip-flops connected to that clock. These period constraints then cover most of the logic paths used in a synchronous design. Here are the clock period constraints used by the Framework Logic:

```
NET "eclk_in" TNM_NET = "eclk_in";
TIMESPEC "TS_eclk_in" = PERIOD "eclk_in" 13 ns HIGH 50 %;
NET "fpdp_tx_clk" TNM_NET = "fpdp_tx_clk";
TIMESPEC "TS_fpdp_tx_clk" = PERIOD "fpdp_tx_clk" 20 ns HIGH 50 %;
NET "Xin_s" TNM_NET = "Xin_s";
TIMESPEC "TS_Xin_s" = PERIOD "Xin_s" 271 ns HIGH 50 %;
NET "fpdp_rx_p_strobe_i" TNM_NET = "fpdp_rx_p_strobe_i";
TIMESPEC "TS_fpdp_rx_p_strobe_i" = PERIOD "fpdp_rx_p_strobe_i" 13 ns HIGH 50 %;
```

Figure 26. Clock Period Constraints.

Figure 19. shows that EMIF clock (eclk_in) is constrained to 13 ns, giving a small margin for a 75MHz bus. External devices require an additional constraint to be sure that we get the signal on- chip and to its destination in time. Since the external chip, such as the DSP, may have a delay from the clock edge to when we get the signal, an additional constraint is defines the amount of time after the clock that the signal is given to the logic. This type of constraint is used on the DSP control signals such as CE, ARE, AWE and addresses to guarantee that setup timings are met. a Timing Group is defined for these signals with a timing constraint for the group.

```
NET "are_n" OFFSET = IN 6 ns BEFORE "eclk_in"  ;
NET "awe_n" OFFSET = IN 6 ns BEFORE "eclk_in"  ;
NET "aoe_n" OFFSET = IN 6 ns BEFORE "eclk_in"  ;
```

```
NET "ce2_n" OFFSET = IN 6 ns BEFORE "eclk_in"  ;
NET "ce3_n" OFFSET = IN 6 ns BEFORE "eclk_in"  ;
```

Figure 27. DSP Signals Timing Constraints.

IO Standard Constraints

Each pin is defined for its appropriate IO standard. The Xilinx default standard is LVTTL, so it is common to leave those out and define only the exceptions. Do not change the IO standard as defined as this may cause damage to the devices.

```
NET "ardy"  LOC = "D5" | IOSTANDARD = LVTTL ;
NET "are_n"  LOC = "C5" | IOSTANDARD = LVTTL ;
NET "awe_n"  LOC = "D7" | IOSTANDARD = LVTTL ;
NET "ce1_n"  LOC = "B6" | IOSTANDARD = LVTTL ;
NET "ce2_n"  LOC = "C6" | IOSTANDARD = LVTTL ;
NET "ce3_n"  LOC = "C4" | IOSTANDARD = LVTTL ;
```

Figure 28. IO Standard Constraint.

Pin Constraints

Each pin has a placement defined in the UCF file, as required by the circuit board design. DO NOT CHANGE these assignments as damage may occur to the Quixote! They must be used on all compiles.

```
NET "eclk_in"  LOC = "C11" | IOSTANDARD = LVTTL ;
```

Pin Assignments For Spartan IIE (300K-600K)

Table 14. I/O Standard - LVTTL

Signal Name	Assigned Pin	Input/ Output	Description
aoe_n (active low)	B5	Input	Asynchronous memory output enable.
app_boot	E15	Input	Application Boot Header.
ardy (active low)	D5	Output	Asynchronous memory read input (Logic to DSP)
are_n (active low)	C5	Input	Asynchronous memory read enable (from DSP).
awe_n (active low)	D7	Input	Asynchronous memory write enable (from DSP).
ce1_n (active low)	B6	Input	Memory space enable (from DSP).
ce2_n (active low)	C6	Input	Memory space enable (from DSP).
ce3_n (active low)	C4	Input	Memory space enable (from DSP).
clk_to_dds	W3	Output	Clock to DDS (80 MHz).
clklink_in	M22	Input	Clock link (high speed signals,usually card to card).
clklink_out	L20	Output	Clock link (high speed signals,usually card to card).

Signal Name	Assigned Pin	Input/ Output	Description
cts_n (active low)	N22	Input	Clear to send (Uart control signal).
dds_fq_ud	U3	Output	DDS control signal.
dds_i	C16	Input	DDS Input.
dds_reset	V3	Output	DDS Reset.
dds_wr_n (active low)	U4	Output	DDS Write.
dio(0) (PULL UP)	U21	InOut	Digital I/O signal.
dio(1) (PULL UP)	V21	InOut	Digital I/O signal.
dio(2) (PULL UP)	W22	InOut	Digital I/O signal.
dio(3) (PULL UP)	V22	InOut	Digital I/O signal.
dio(4) (PULL UP)	R21	InOut	Digital I/O signal.
dio(5) (PULL UP)	T21	InOut	Digital I/O signal.
dio(6) (PULL UP)	U20	InOut	Digital I/O signal.
dio(7) (PULL UP)	AB18	InOut	Digital I/O signal.
dio(8) (PULL UP)	T19	InOut	Digital I/O signal.
dio(9) (PULL UP)	U19	InOut	Digital I/O signal.
dio(10) (PULL UP)	Y19	InOut	Digital I/O signal.
dio(11) (PULL UP)	T20	InOut	Digital I/O signal.
dio(12) (PULL UP)	N18	InOut	Digital I/O signal.
dio(13) (PULL UP)	P18	InOut	Digital I/O signal.
dio(14) (PULL UP)	R18	InOut	Digital I/O signal.
dio(15) (PULL UP)	T18	InOut	Digital I/O signal.
dio(16) (PULL UP)	M17	InOut	Digital I/O signal.
dio(17) (PULL UP)	P17	InOut	Digital I/O signal.
dio(18) (PULL UP)	L18	InOut	Digital I/O signal.
dio(19) (PULL UP)	M18	InOut	Digital I/O signal.
dio(20) (PULL UP)	N19	InOut	Digital I/O signal.
dio(21) (PULL UP)	P19	InOut	Digital I/O signal.
dio(22) (PULL UP)	M19	InOut	Digital I/O signal.
dio(23) (PULL UP)	R20	InOut	Digital I/O signal.
dio(24) (PULL UP)	M20	InOut	Digital I/O signal.
dio(25) (PULL UP)	N20	InOut	Digital I/O signal.
dio(26) (PULL UP)	P20	InOut	Digital I/O signal.
dio(27) (PULL UP)	P21	InOut	Digital I/O signal.
dio(28) (PULL UP)	M21	InOut	Digital I/O signal.
dio(29) (PULL UP)	N21	InOut	Digital I/O signal.
dio(30) (PULL UP)	L21	InOut	Digital I/O signal.
dio(31) (PULL UP)	K22	InOut	Digital I/O signal.
dsp_dma_int	E16	Output	DSP DMA Interrupt.
dsp_int(4)	K6	Output	Interrupt to DSP.
dsp_int(5)	L6	Output	Interrupt to DSP.

Signal Name	Assigned Pin	Input/ Output	Description
dsp_int(6)	K5	Output	Interrupt to DSP.
dsp_int(7)	L5	Output	Interrupt to DSP.
dsp_rd_fifo_int	AB3	Input	Interrupt from Inter-processor FIFO (Spartan IIe 50K)
dsp_reset	J18	Output	Reset to DSP.
dsp_t0	A15	Input	DSP Timer0.
dsp_t1	B15	Input	DSP Timer1.
dsp_wr_fifo_int	AA3	Input	Interrupt from Inter-processor FIFO (Spartan IIe 50K)
dtr	L22	Output	Data terminal ready (from UART).
ea(2)	B3	Input	EMIF ADDRESS
ea(3)	E8	Input	EMIF ADDRESS
ea(4)	A5	Input	EMIF ADDRESS
ea(5)	D8	Input	EMIF ADDRESS
ea(6)	E7	Input	EMIF ADDRESS
ea(7)	B4	Input	EMIF ADDRESS
ea(8)	A4	Input	EMIF ADDRESS
ea(9)	C8	Input	EMIF ADDRESS
ea(10)	C7	Input	EMIF ADDRESS
ea(11)	D6	Input	EMIF ADDRESS
ea(12)	A3	Input	EMIF ADDRESS
ea(13)	B7	Input	EMIF ADDRESS
ea(14)	A7	Input	EMIF ADDRESS
ea(15)	A6	Input	EMIF ADDRESS
ea(16)	A8	Input	EMIF ADDRESS
ea(17)	B8	Input	EMIF ADDRESS
ea(18)	B9	Input	EMIF ADDRESS
ea(19)	B10	Input	EMIF ADDRESS
ea(20)	A10	Input	EMIF ADDRESS
ea(21)	A9	Input	EMIF ADDRESS
eclk_in	C11	Input	External EMIF input clock.
ed(0)	C9	InOut	EMIF DATA
ed(1)	D9	InOut	EMIF DATA
ed(2)	E9	InOut	EMIF DATA
ed(3)	F9	InOut	EMIF DATA
ed(4)	C10	InOut	EMIF DATA
ed(5)	D10	InOut	EMIF DATA
ed(6)	E10	InOut	EMIF DATA
ed(7)	F10	InOut	EMIF DATA
ed(8)	D11	InOut	EMIF DATA
ed(9)	E11	InOut	EMIF DATA

Signal Name	Assigned Pin	Input/ Output	Description
ed(10)	F11	InOut	EMIF DATA
ed(11)	B11	InOut	EMIF DATA
ed(12)	A12	InOut	EMIF DATA
ed(13)	B12	InOut	EMIF DATA
ed(14)	C12	InOut	EMIF DATA
ed(15)	D12	InOut	EMIF DATA
ed(16)	E12	InOut	EMIF DATA
ed(17)	F12	InOut	EMIF DATA
ed(18)	A13	InOut	EMIF DATA
ed(19)	B13	InOut	EMIF DATA
ed(20)	C13	InOut	EMIF DATA
ed(21)	D13	InOut	EMIF DATA
ed(22)	E13	InOut	EMIF DATA
ed(23)	F13	InOut	EMIF DATA
ed(24)	A14	InOut	EMIF DATA
ed(25)	B14	InOut	EMIF DATA
ed(26)	C14	InOut	EMIF DATA
ed(27)	D14	InOut	EMIF DATA
ed(28)	E14	InOut	EMIF DATA
ed(29)	F14	InOut	EMIF DATA
ed(30)	C15	InOut	EMIF DATA
ed(31)	D15	InOut	EMIF DATA
ext_clk	D17	Input	External Clock
ext_dig_trig (PULL UP)	H22	Input	External Digital Trigger.
ext_int_in	D16	Input	External Interrupt.
fpdp_rx_din(0)	AB10	Input	FPDP Receiver Data
fpdp_rx_din(1)	AA10	Input	FPDP Receiver Data
fpdp_rx_din(2)	Y10	Input	FPDP Receiver Data
fpdp_rx_din(3)	W10	Input	FPDP Receiver Data
fpdp_rx_din(4)	V10	Input	FPDP Receiver Data
fpdp_rx_din(5)	AB9	Input	FPDP Receiver Data
fpdp_rx_din(6)	AA9	Input	FPDP Receiver Data
fpdp_rx_din(7)	Y9	Input	FPDP Receiver Data
fpdp_rx_din(8)	W9	Input	FPDP Receiver Data
fpdp_rx_din(9)	V9	Input	FPDP Receiver Data
fpdp_rx_din(10)	AB8	Input	FPDP Receiver Data
fpdp_rx_din(11)	AA8	Input	FPDP Receiver Data
fpdp_rx_din(12)	Y8	Input	FPDP Receiver Data
fpdp_rx_din(13)	W8	Input	FPDP Receiver Data
fpdp_rx_din(14)	V8	Input	FPDP Receiver Data

Signal Name	Assigned Pin	Input/ Output	Description
fpdp_rx_din(15)	AB7	Input	FPDP Receiver Data
fpdp_rx_din(16)	AA7	Input	FPDP Receiver Data
fpdp_rx_din(17)	Y7	Input	FPDP Receiver Data
fpdp_rx_din(18)	W7	Input	FPDP Receiver Data
fpdp_rx_din(19)	V7	Input	FPDP Receiver Data
fpdp_rx_din(20)	AB6	Input	FPDP Receiver Data
fpdp_rx_din(21)	AA6	Input	FPDP Receiver Data
fpdp_rx_din(22)	Y6	Input	FPDP Receiver Data
fpdp_rx_din(23)	W6	Input	FPDP Receiver Data
fpdp_rx_din(24)	V6	Input	FPDP Receiver Data
fpdp_rx_din(25)	AB5	Input	FPDP Receiver Data
fpdp_rx_din(26)	AA5	Input	FPDP Receiver Data
fpdp_rx_din(27)	Y5	Input	FPDP Receiver Data
fpdp_rx_din(28)	W5	Input	FPDP Receiver Data
fpdp_rx_din(29)	AB4	Input	FPDP Receiver Data
fpdp_rx_din(30)	AB11	Input	FPDP Receiver Data
fpdp_rx_din(31)	AA11	Input	FPDP Receiver Data
fpdp_rx_dvalid_n(active low)	W11	Input	FPDP Rx Data Valid (i.e. Data is Valid from FPDP tx)
fpdp_rx_nrdy_n	U10	Output	Rx not ready until the FPDP reset is deasserted.
fpdp_rx_p_strobe_i	A11	Input	FPDP Rx Strobe
fpdp_rx_pio(0)	V11	InOut	The PIO signals are programmable I/O lines.
fpdp_rx_pio(1)	U11	InOut	The PIO signals are programmable I/O lines.
fpdp_rx_suspend_n	Y11	Output	This signal suspends the data flow when the Rx FIFO is nearly full
fpdp_rx_sync_n	U9	Input	FPDP Rx receives sync_n to begin.
fpdp_tx_dir_n	V12	Output	FPDP Tx asserts direction to low.
fpdp_tx_do<0>	Y12	Output	FPDP Tx transmit data.
fpdp_tx_do<1>	W12	Output	FPDP Tx transmit data.
fpdp_tx_do<2>	AA16	Output	FPDP Tx transmit data.
fpdp_tx_do<3>	Y16	Output	FPDP Tx transmit data.
fpdp_tx_do<4>	AB13	Output	FPDP Tx transmit data.
fpdp_tx_do<5>	AA13	Output	FPDP Tx transmit data.
fpdp_tx_do<6>	Y13	Output	FPDP Tx transmit data.
fpdp_tx_do<7>	W13	Output	FPDP Tx transmit data.
fpdp_tx_do<8>	V13	Output	FPDP Tx transmit data.
fpdp_tx_do<9>	U13	Output	FPDP Tx transmit data.
fpdp_tx_do<10>	AB14	Output	FPDP Tx transmit data.
fpdp_tx_do<11>	AA14	Output	FPDP Tx transmit data.
fpdp_tx_do<12>	Y14	Output	FPDP Tx transmit data.
fpdp_tx_do<13>	W14	Output	FPDP Tx transmit data.

Signal Name	Assigned Pin	Input/ Output	Description
fpdp_tx_do<14>	V14	Output	FPDP Tx transmit data.
fpdp_tx_do<15>	U14	Output	FPDP Tx transmit data.
fpdp_tx_do<16>	AB15	Output	FPDP Tx transmit data.
fpdp_tx_do<17>	AA15	Output	FPDP Tx transmit data.
fpdp_tx_do<18>	Y15	Output	FPDP Tx transmit data.
fpdp_tx_do<19>	W15	Output	FPDP Tx transmit data.
fpdp_tx_do<20>	V15	Output	FPDP Tx transmit data.
fpdp_tx_do<21>	AB16	Output	FPDP Tx transmit data.
fpdp_tx_do<22>	W16	Output	FPDP Tx transmit data.
fpdp_tx_do<23>	V16	Output	FPDP Tx transmit data.
fpdp_tx_do<24>	AB20	Output	FPDP Tx transmit data.
fpdp_tx_do<25>	AA20	Output	FPDP Tx transmit data.
fpdp_tx_do<26>	AB17	Output	FPDP Tx transmit data.
fpdp_tx_do<27>	AA18	Output	FPDP Tx transmit data.
fpdp_tx_do<28>	Y18	Output	FPDP Tx transmit data.
fpdp_tx_do<29>	W17	Output	FPDP Tx transmit data.
fpdp_tx_do<30>	V17	Output	FPDP Tx transmit data.
fpdp_tx_do<31>	W18	Output	FPDP Tx transmit data.
fpdp_tx_dvalid_n	U18	Output	Data Valid assertion indicates that data bus has valid data.
fpdp_tx_n_strobe (IOSTANDARD LVPECL)	AA17	Output	This Strobe is the negative version of the differential PECL data strobe.
fpdp_tx_nrdy_n	AB19		
fpdp_tx_p_strobe (IOSTANDARD LVPECL)	Y17	Output	This Strobe is the positive version of the differential PECL data strobe.
fpdp_tx_pio<0>	V19	InOut	The PIO signals are programmable I/O lines.
fpdp_tx_pio<1>	V20	InOut	The PIO signals are programmable I/O lines.
fpdp_tx_suspend_n (active low)	AA19	Input	This signal suspends the data flow when the Rx FIFO is nearly full
fpdp_tx_sync_n (active low)	U12	Output	FPDP Tx provides a sync pulse to synchronizing data transfers in different modes.
iomod_n<0>	C1	Output	Omnibus Strobe 0.
iomod_n<1>	C2	Output	Omnibus Strobe 1.
iomod_n<2>	D1	Output	Omnibus Strobe 2.
iomod_n<3>	D2	Output	Omnibus Strobe 3.
iomod_n<4>	D3	Output	Omnibus Strobe 4.
iomod_n<5>	E1	Output	Omnibus Strobe 5.
iomod_n<6>	E2	Output	Omnibus Strobe 6.
iomod_n<7>	E3	Output	Omnibus Strobe 7.

Signal Name	Assigned Pin	Input/ Output	Description
iordy_n<0> (active low)	F1	Input	Ready Signal from module 0.
iordy_n<1> (active low)	F2	Input	Ready Signal from module 1.
mod0_data<0>	T3	InOut	Module 0 Data
mod0_data<1>	R3	InOut	Module 0 Data
mod0_data<2>	P3	InOut	Module 0 Data
mod0_data<3>	N3	InOut	Module 0 Data
mod0_data<4>	M3	InOut	Module 0 Data
mod0_data<5>	L3	InOut	Module 0 Data
mod0_data<6>	K3	InOut	Module 0 Data
mod0_data<7>	J3	InOut	Module 0 Data
mod0_data<8>	Y2	InOut	Module 0 Data
mod0_data<9>	W2	InOut	Module 0 Data
mod0_data<10>	V2	InOut	Module 0 Data
mod0_data<11>	U2	InOut	Module 0 Data
mod0_data<12>	T2	InOut	Module 0 Data
mod0_data<13>	R2	InOut	Module 0 Data
mod0_data<14>	P2	InOut	Module 0 Data
mod0_data<15>	N2	InOut	Module 0 Data
mod0_data<16>	M2	InOut	Module 0 Data
mod0_data<17>	L2	InOut	Module 0 Data
mod0_data<18>	K2	InOut	Module 0 Data
mod0_data<19>	J2	InOut	Module 0 Data
mod0_data<20>	Y1	InOut	Module 0 Data
mod0_data<21>	W1	InOut	Module 0 Data
mod0_data<22>	V1	InOut	Module 0 Data
mod0_data<23>	U1	InOut	Module 0 Data
mod0_data<24>	T1	InOut	Module 0 Data
mod0_data<25>	R1	InOut	Module 0 Data
mod0_data<26>	P1	InOut	Module 0 Data
mod0_data<27>	N1	InOut	Module 0 Data
mod0_data<28>	M1	InOut	Module 0 Data
mod0_data<29>	L1	InOut	Module 0 Data
mod0_data<30>	K1	InOut	Module 0 Data
mod0_data<31>	J1	InOut	Module 0 Data
mod0_int<0>	F4	Input	Interrupt 0 from module 0.
mod0_int<1>	F5	Input	Interrupt 1 from module 0.
mod0_timer<0>	J5	Output	Module 0 timer 0, clocks output selection
mod0_timer<1>	H3	Output	Module 0 timer 1, clocks output selection.
mod1_data<0>	J22	InOut	Module 1 Data
mod1_data<1>	K20	InOut	Module 1 Data

Signal Name	Assigned Pin	Input/ Output	Description
mod1_data<2>	J20	InOut	Module 1 Data
mod1_data<3>	G22	InOut	Module 1 Data
mod1_data<4>	F22	InOut	Module 1 Data
mod1_data<5>	E22	InOut	Module 1 Data
mod1_data<6>	D22	InOut	Module 1 Data
mod1_data<7>	K21	InOut	Module 1 Data
mod1_data<8>	J21	InOut	Module 1 Data
mod1_data<9>	H21	InOut	Module 1 Data
mod1_data<10>	G21	InOut	Module 1 Data
mod1_data<11>	F21	InOut	Module 1 Data
mod1_data<12>	E21	InOut	Module 1 Data
mod1_data<13>	D21	InOut	Module 1 Data
mod1_data<14>	G20	InOut	Module 1 Data
mod1_data<15>	F20	InOut	Module 1 Data
mod1_data<16>	E20	InOut	Module 1 Data
mod1_data<17>	D20	InOut	Module 1 Data
mod1_data<18>	J19	InOut	Module 1 Data
mod1_data<19>	G19	InOut	Module 1 Data
mod1_data<20>	L19	InOut	Module 1 Data
mod1_data<21>	K18	InOut	Module 1 Data
mod1_data<22>	L17	InOut	Module 1 Data
mod1_data<23>	J17	InOut	Module 1 Data
mod1_data<24>	H18	InOut	Module 1 Data
mod1_data<25>	G18	InOut	Module 1 Data
mod1_data<26>	F18	InOut	Module 1 Data
mod1_data<27>	D18	InOut	Module 1 Data
mod1_data<28>	C18	InOut	Module 1 Data
mod1_data<29>	B18	InOut	Module 1 Data
mod1_data<30>	C17	InOut	Module 1 Data
mod1_data<31>	B17	InOut	Module 1 Data
mod1_int<0>	G2	Input	Interrupt 0 from module 1.
mod1_int<1>	H4	Input	Interrupt 1 from module 1.
mod1_timer<0>	A16	Output	Module 1 timer 0, clocks output selection
mod1_timer<1>	B16	Output	Module 1 timer 1, clocks output selection.
mod_addr<2>	M6	Output	Omnibus Module Address.
mod_addr<3>	N6	Output	Omnibus Module Address.
mod_addr<4>	P6	Output	Omnibus Module Address.
mod_addr<5>	M5	Output	Omnibus Module Address.
mod_addr<6>	N5	Output	Omnibus Module Address.
mod_addr<7>	P5	Output	Omnibus Module Address.

Signal Name	Assigned Pin	Input/ Output	Description
mod_addr<8>	R5	Output	Omnibus Module Address.
mod_addr<9>	T5	Output	Omnibus Module Address.
mod_addr<10>	N4	Output	Omnibus Module Address.
mod_addr<11>	P4	Output	Omnibus Module Address.
mod_addr<12>	R4	Output	Omnibus Module Address.
mod_addr<13>	T4	Output	Omnibus Module Address.
mod_clk<0>	H1	Output	Module 0 Clock
mod_clk<1>	G1	Output	Module 1 Clock.
mod_dds<0>	F3	Output	Module 0 DDS
mod_dds<1>	G4	Output	Module 1 DDS
mod_rw	M4	Output	Read (active high) and Write (active low) to module.
mod_trig_n<0>	G5	Input	Module Trigger 0
mod_trig_n<1>	G3	Input	Module Trigger 1
mod_trig_n<2>	H2	Input	Module Trigger 2
mod_trig_n<3>	H5	Input	Module Trigger 3
nmi	J6	Output	Interrupt to DSP
reset_n (active low)	K17	Input	Reset to the Logic.
rts_n (active low)	U22	Output	Request to send (UART).
rxd	T22	Input	Receive Data (UART)
synclink_extimer_t<0> (PULL UP)	F19	InOut	Synclink - Multi-Card Timing Synchronization Counters - Run through external inputs, readable through DSP.
synclink_extimer_t<1> (PULL UP)	E19	InOut	Synclink - Multi-Card Timing Synchronization Counters - Run through external inputs, readable through DSP.
synclink_extimer_t<2> (PULL UP)	E17	InOut	Synclink - Multi-Card Timing Synchronization Counters - Run through external inputs, readable through DSP.
synclink_extimer_t<3> (PULL UP)	B19	InOut	Synclink - Multi-Card Timing Synchronization Counters - Run through external inputs, readable through DSP.
synclink_extimer_t<4> (PULL UP)	A19	InOut	Synclink - Multi-Card Timing Synchronization Counters - Run through external inputs, readable through DSP.
synclink_extimer_t<5> (PULL UP)	A18	InOut	Synclink - Multi-Card Timing Synchronization Counters - Run through external inputs, readable through DSP.
timebase	A17	Input	80.00MHz
tp<0>	K4	Output	Test Point 0.
tp<1>	J4	Output	Test Point 1.
tp<2>	L4	Output	Test Point 2.

Signal Name	Assigned Pin	Input/ Output	Description
txd	R22	Output	Transmit Data.
Xin	AB21	Input	1.8432 MHz to UART.
CONFIG PROHIBIT	C21		Prohibit Contraint
CONFIG PROHIBIT	C22		Prohibit Contraint
CONFIG PROHIBIT	H19		Prohibit Contraint
CONFIG PROHIBIT	H20		Prohibit Contraint
CONFIG PROHIBIT	K19		Prohibit Contraint
CONFIG PROHIBIT	N17		Prohibit Contraint
CONFIG PROHIBIT	P22		Prohibit Contraint
CONFIG PROHIBIT	R19		Prohibit Contraint
CONFIG PROHIBIT	Y22		Prohibit Contraint
CONFIG PROHIBIT	W21		Prohibit Contraint
CONFIG PROHIBIT	A20		Prohibit Contraint
CONFIG PROHIBIT	B20		Prohibit Contraint

Simulation

The test files are used in the simulation and testing of the Framework code. The testbench file is SBC6713_top_tb.vhd and it uses several components for testing that are defined by the other model files. These model files are very simple and are only for simple testing. More complex models may be needed to adequately model more advanced uses.

The testbench contains a set of simulation steps that exercise various functions on the framework logic for basic interface testing. Behavioral procedures have been written to simulate the DSP timing for sync and async memory accesses that are useful in simulating data movement. Also, the steps to setup the logic for data streaming support are shown so that interrupt servicing (DMA or CPU accesses), trigger and event log use are illustrated.

Required Software and Hardware

Recommended Software: Mentor Graphics ModelSim 5.7e or higher

Recommended Hardware: Pentium 3 or better at 500 MHz with 512M of RAM

Setting Up the Simulation

The files unzipped from the Framework Logic archive contain all the source and macro files needed. You will normally need to make a ModelSim project reflecting your exact directory structure, although an mpf file is provided. You will also need to compile the Xilinx unisim, simprim and core libs and point to them in ModelSim. Set the project default to VHDL '93 if you intend to compile in ModelSim using the project manager.

Loading the Testbench

The simulator used from within ISE is Mentor Graphics ModelSim and support files for using the testbench from within environment are included. We recommend that you run ModelSim using the macro files provided as a standalone operation, and not from within ISE. This is because we have several models that do not get compiled if you run from ISE.

The macro files are the tb_sbc6713.do and wave.do files for compilation and waveform visualization, respectively. The tb_sbc6713.do file compiles the files in the necessary order and loads the simulation.

Running the Simulations

Once you have executed the macro files in ModelSim, the simulation is ready to run. A time period of 100 us is usually enough to see it start.

Modifying the Simulations

The testbench file provides sample code to begin your simulations. Accesses are shown to each peripheral from the DSP, and how to control the peripherals such as Omnibus, FPDP,UART, etc..In the code are several procedures that simulate DSP accesses and can be used to simulate both async and burst DSP access to the logic. The following code shows the DSP async access procedure, with the correct timing, and its use in the testbench. Below is an example for Omnibus with Asynchronous accesses.

```
procedure async_access (
        constant ce_space : integer;
        constant rw : std_logic;
        constant address : bit_vector(23 downto 0);
        constant data: bit_vector(31 downto 0)
        ) is
  begin
        wait until eclk_in'event and eclk_in = '1';
        case ce_space is
                when 0 =>
                        ce0_n <= '0' after 2 ns;
                        ce1_n <= '1';
                        ce2_n <= '1';
                        ce3_n <= '1';
                when 1 =>
                        ce0_n <= '1';
                        ce1_n <= '0' after 2 ns;
                        ce2_n <= '1';
                        ce3_n <= '1';
                when 2 =>
                        ce0_n <= '1';
                        ce1_n <= '1';
                        ce2_n <= '0' after 2 ns;
                        ce3_n <= '1';
                when 3 =>
                        ce0_n <= '1';
                        ce1_n <= '1';
                        ce2_n <= '1';
                        ce3_n <= '0' after 2 ns;
                when others =>
                        ce0_n <= '1';
                        ce1_n <= '1';
                        ce2_n <= '1';
                        ce3_n <= '1';
        end case;
        -- this is a setup of one cycle
        if (rw = '1') then
                aoe_n <= '0' after 2 ns;
        else
                aoe_n <= '1';
        end if;
        ea <= to_stdlogicvector(address(21 downto 2)) after 2 ns;
        if (rw = '0') then
                ed <= (to_stdlogicvector(data)) after 2 ns;
        else
                ed <= (others => 'H');
        end if;
        wait until eclk_in'event and eclk_in = '1';
        -- one cycle access
        if (rw = '1') then
                are_n <= '0' after 2 ns;
        else
```

```
                awe_n <= '0' after 2 ns;
        end if;

        wait until eclk_in'event and eclk_in = '1';
        wait until eclk_in'event and eclk_in = '1';
        if ( ce_space = 2 ) then
                printval(ed);          -- sample point from dsp
                wait until eclk_in'event and eclk_in = '1';
        end if;

xxx:    while ((ardy = '0' or ardy = 'L') and ce0_n /= '0') loop
                wait until eclk_in'event and eclk_in = '1';
        end loop;

        wait until eclk_in'event and eclk_in = '1';
        are_n <= '1';
        awe_n <= '1';

        wait until eclk_in'event and eclk_in = '1';
        aoe_n <= '1' after 2 ns;
        ce0_n <= '1';
        ce1_n <= '1';
        ce2_n <= '1';
        ce3_n <= '1';
        ea <= (others => 'H') after 2 ns;
        ed <= (others => 'H') after 2 ns;
        gap;
end async_access;
```

Figure 29. VHDL Procedure for Simulating a DSP Async Access.

```
-- module 0 tests

 print(" ");
 print("MODULE 0 TEST");

-- write module 0, iomod1 in async mode
async_access(3, '0', X"010000", X"12345678");

-- read module 0, iomod1 in async mode
async_access(3, '1', X"010000", X"00000000");

-- write module 0, iomod1 in async mode
async_access(3, '0', X"010000", X"87654321");

-- read module 0, iomod1 in async mode
async_access(3, '1', X"010000", X"00000000");

-- write module 0, iomod1 in async mode
async_access(3, '0', X"010000", X"09876543");

-- read module 0, iomod1 in async mode
async_access(3, '1', X"010000", X"00000000");

-- write module 0, iomod1 in async mode
async_access(3, '0', X"010000", X"345678AB");

-- read module 0, iomod1 in async mode
async_access(3, '1', X"010000", X"00000000");
```

Figure 30. VHDL Testbench Code For Omnibus.

Omnibus Interface

Omnibus is an open standard that allows customers to design application specific modules to use with SBC6713e. The Omnibus slots are accessed as memory-mapped peripherals with the SBC6713e providing four decoded chip select signals per slot, for a total of eight on SBC6713e. Please refer to table 15 for SBC6713e I/O bus memory mapping. Each module site provides 32-bit wide data bus connection to the processor's data bus with 12-bit of low order address signals for additional decoding.

Omnibus accesses are synchronous to a single clock (MOD_CLK) and start and stop on the falling edges of that clock. All accesses are defined by activity on the IOMOD_n(x) decoded chip select signals. As we can see in figure 23, an access is active when one of the IOMOD_n(x) signals is low. The bus is completely inactive when all IOMOD_n(x) signals to all available omnibus sites are high. The minimum bus access length is two clocks, and there is a one clock dwell time between accesses. The IORDY_n(x) (active low ready) is generated by Omnibus modules which are responsible to terminate bus accesses to their respective memory mapped areas. This allows each module to individually determine timing for bus accesses to the memory space in which it is installed. MOD_RW is read and write signal where read is active high, and write is active low. For more information about Omnibus Modules please go to (www.innovative-dsp.com) |Support |Application Notes| OMNIBUS Modules Or (http://www.innovative-dsp.com/support/appnotes.htm).

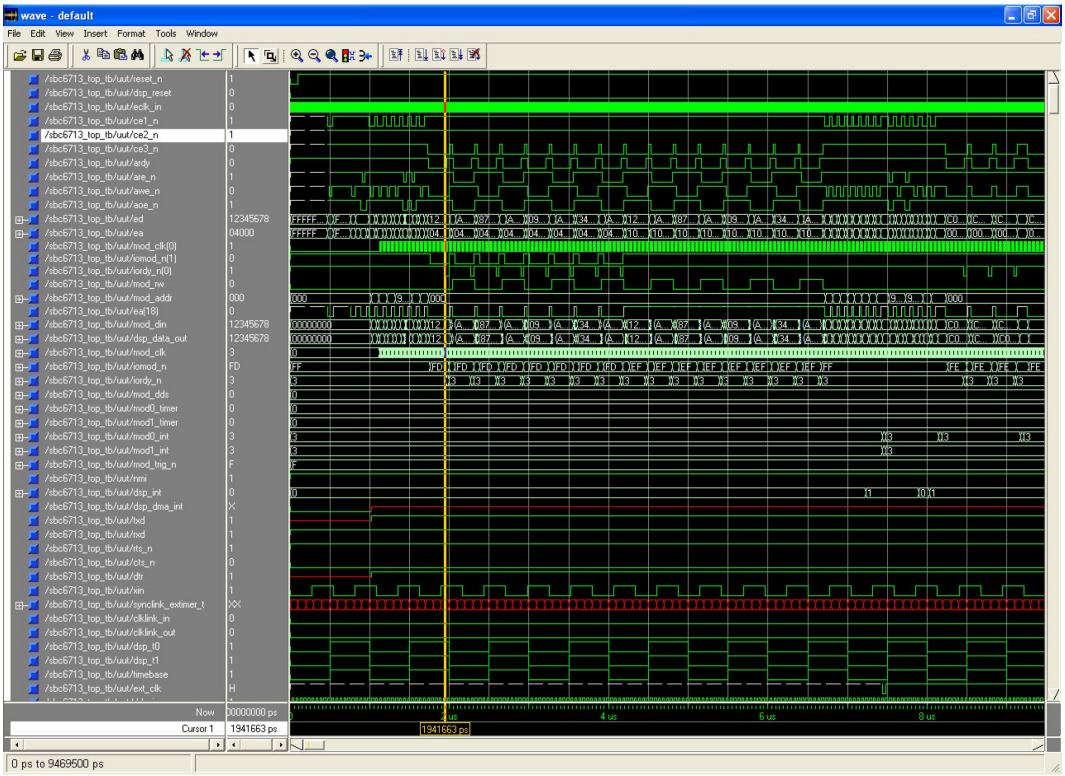


Figure 31. Omnibus Writes And Reads.

Burst Reads And Writes

There are also Testbench Procedures for Burst Accesses.Bellow is the example code for Burst Reads. This sample code gives you an access to program different burst counts, 32-bit data words.

```
procedure burst_read (
        constant ce_space : integer;
        constant burst_count : integer;
        constant address : bit_vector(23 downto 0)
        ) is
        variable i : integer;
        variable acnt : std_logic_vector(1 downto 0);
  begin
        wait until eclk_in'event and eclk_in = '1';
        case ce_space is
                when 0 =>
                        ce0_n <= '0' after 2 ns;
                        ce1_n <= '1';
                        ce2_n <= '1';
                        ce3_n <= '1';
                when 1 =>
                        ce0_n <= '1';
                        ce1_n <= '0' after 2 ns;
                        ce2_n <= '1';
                        ce3_n <= '1';
                when 2 =>
                        ce0_n <= '1';
                        ce1_n <= '1';
                        ce2_n <= '0' after 2 ns;
                        ce3_n <= '1';
                when 3 =>
                        ce0_n <= '1';
                        ce1_n <= '1';
                        ce2_n <= '1';
                        ce3_n <= '0' after 2 ns;
                when others =>
                        ce0_n <= '1';
                        ce1_n <= '1';
                        ce2_n <= '1';
                        ce3_n <= '1';
        end case;

        -- address strobe control
        acnt := acnt + 1;
        are_n <= '0' after 2 ns;
        ea <= to_stdlogicvector(address(21 downto 2)) after 2 ns;

        wait until eclk_in'event and eclk_in = '1';
        acnt := acnt + 1;
        are_n <= '1' after 2 ns;
        aoe_n <= '0' after 2 ns;

        acnt := "10";
        for i in 1 to (burst_count - 1) loop
        wait until eclk_in'event and eclk_in = '1';

        if (acnt = "00") then
                are_n <= '0' after 2 ns;
        else
                are_n <= '1' after 2 ns;
        end if;
```

```
            acnt := acnt + 1;

            end loop;

            wait until eclk_in'event and eclk_in = '1';
            ce0_n <= '1';
            ce1_n <= '1';
            ce2_n <= '1';
            ce3_n <= '1';
            ea <= (others => 'H') after 2 ns;

            wait until eclk_in'event and eclk_in = '1';

    --      finish burst write
            are_n <= '1';
            awe_n <= '1';
            aoe_n <= '1';
            ed <= (others => 'H') after 2 ns;
    end burst_read;
```

Figure 32. VHDL Procedure For DSP Burst Read Access.

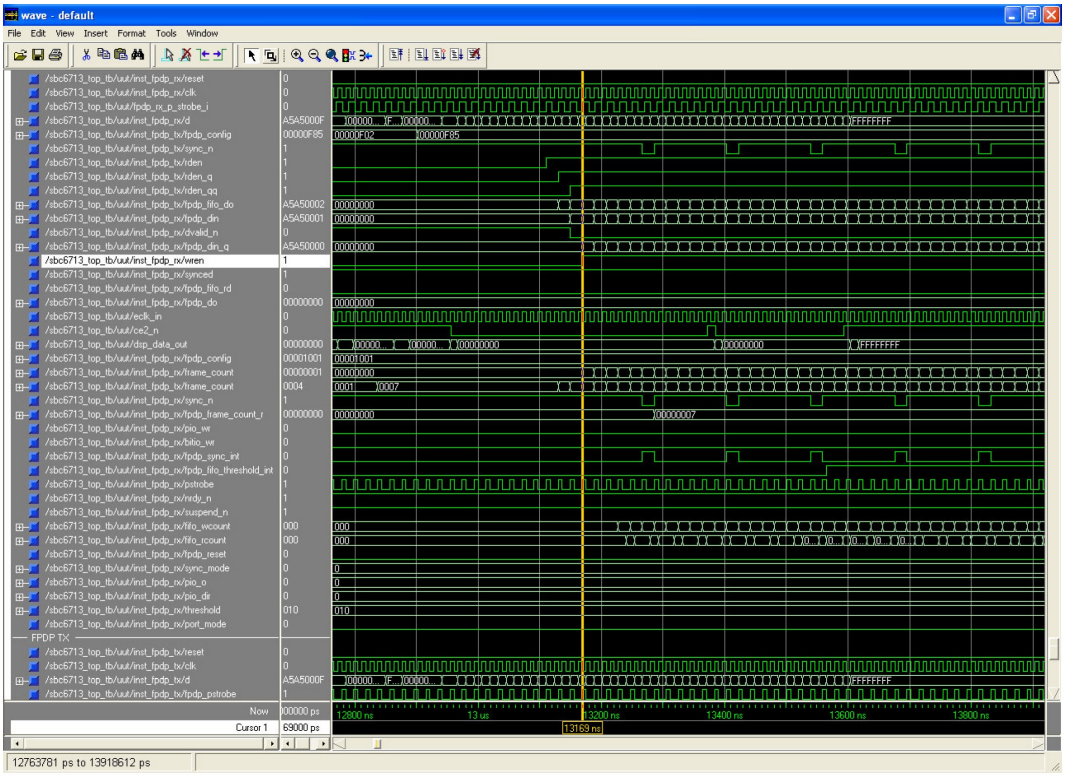


Figure 33. Data Read From FPDP Tx FIFO And Data Write to FPDP Rx FIFO.

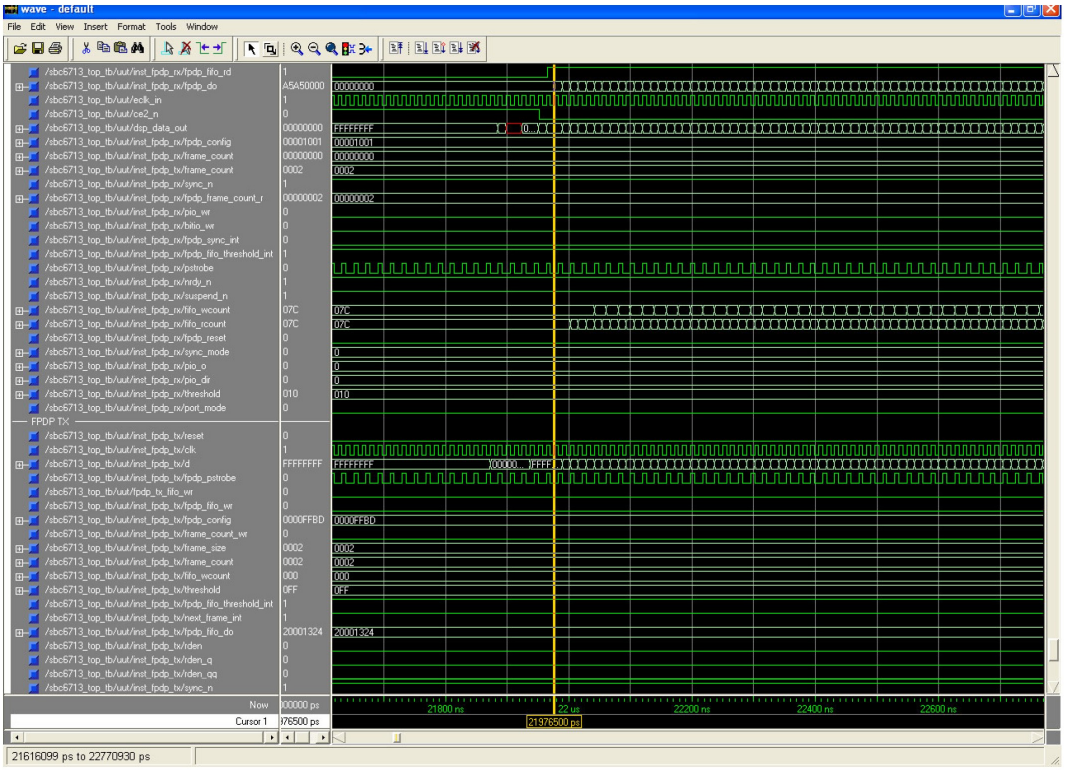


Figure 34. DSP Burst Reads From FPDP Rx FIFO.

```
burst_write(2, 32, FPDP_FIFO, X"a5a50000"); -- write some data into the FPDP FIFO
burst_read (2, 64, FPDP_FIFO); -- Read Data from FPDP Rx FIFO
```

Figure 35. Burst Writes And Reads.

```
print(" ");
print("CHECKING INTERRUPT");


-- set the interrupts polarity and type
async_access(1, '0', X"000400", X"00000000");  -- set all interrupts active low
async_access(1, '0', X"000300", X"00000000");  -- set interrupts to edge sensitive

-- select interrupts
async_access(1, '0', X"000900", X"40100000");  -- nmi int = ext_clk(0)using ack'd int
async_access(1, '0', X"000500", X"C0010000");  -- int4 = mod0 int0 using ack'd int
async_access(1, '0', X"000600", X"00020000");  -- int5 = mod0 int1
async_access(1, '0', X"000700", X"00040000");  -- int6 = mod1 int0
async_access(1, '0', X"000800", X"00000080");  -- int7 = mod1 int1
async_access(1, '0', X"000A00", X"00080000");  -- dma_int =fpdp rx fifo interrupt
```

Figure 36. Sample For Simulating DSP Interrupt Servicing

Loading the Logic Image

For the SBC6713e, usually an EXORMacs format text file must be generated from the output .BIT file produced by the place and route process. This is done by opening the PROM File Formatter utility from within the ISE and converting the BIT file into an EXO file. The PROM properties must be set to “byte- wide” and EXORMacs to generate the required EXO file. Once the properties are set, the .EXO file is generated in the source directory. Please refer to the pictures below:

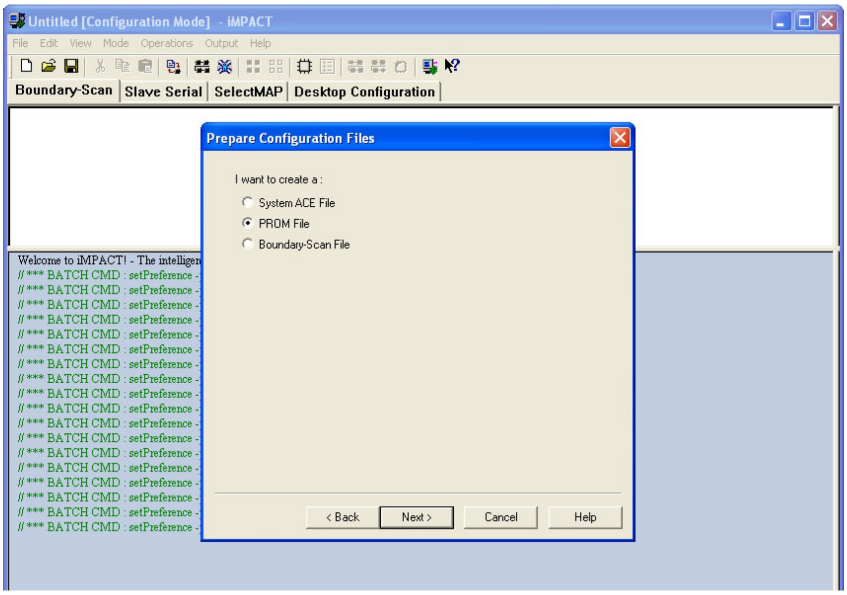


Figure 37. Click on PROM File Option.

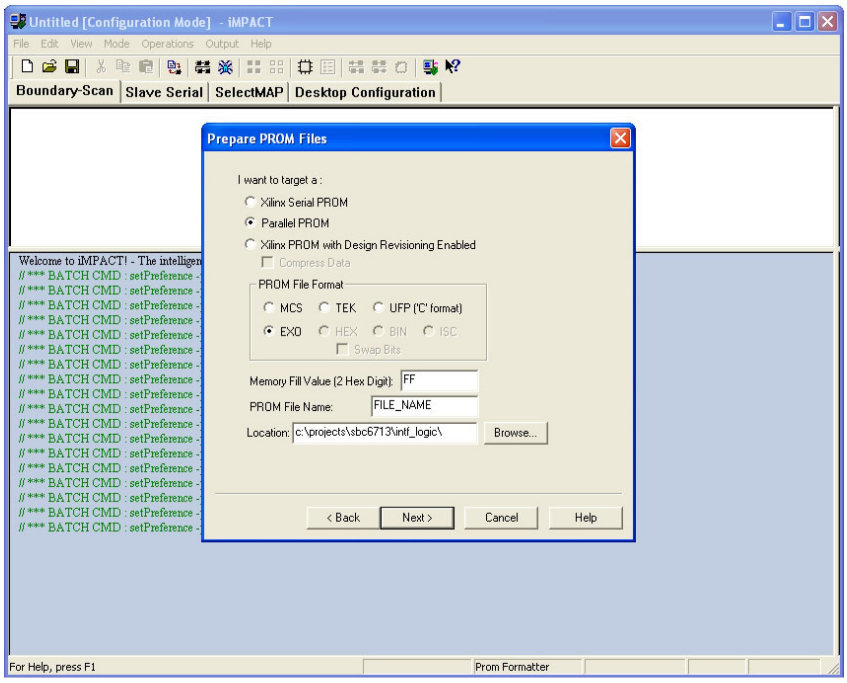


Figure 38. PROM Configuration.

In PROM Configuration:

- Select Parallel PROM.
- Select EXO as File Format.
- Memory Fill Value - 0xFF
- Provide a File Name.

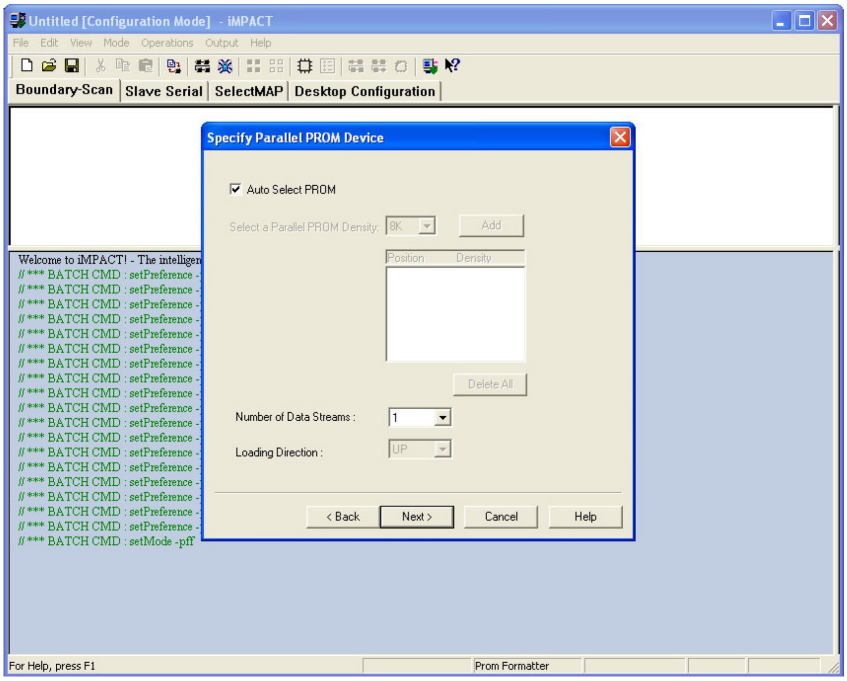


Figure 39. Select "*Auto Select PROM*"

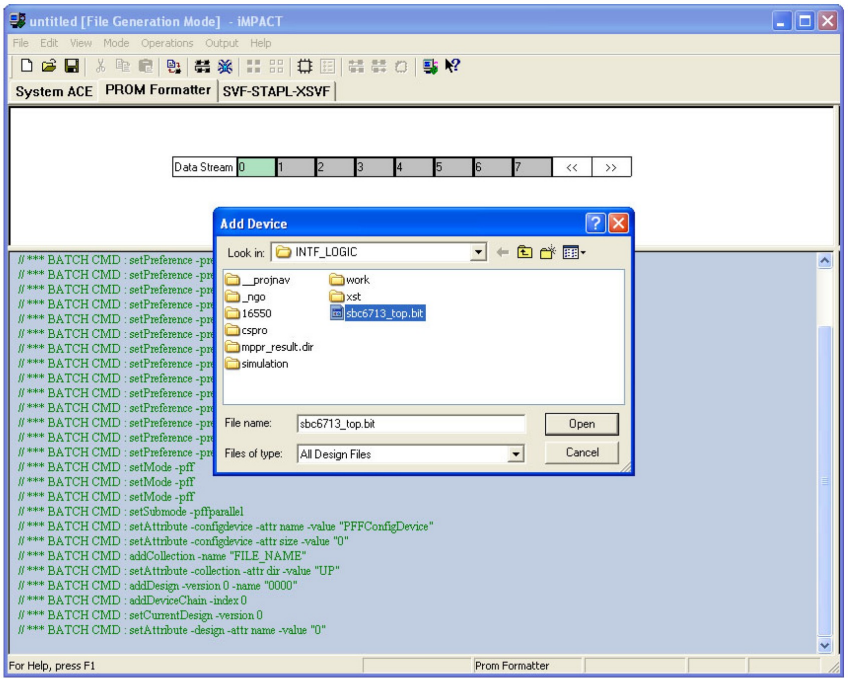

untitled [File Generation Mode] - iMPACT
File Edit View Mode Operations Output Help
System ACE
PROM Formatter
SVF-STAPL-XSVF
Data Stream 0 1 2 3 4 5 6 7 << >>
Add Device
Look in: INTF_LOGIC
__projnav
_ngo
16550
cspro
mppr_result.dir
simulation
work
xst
sbc6713_top.bit
File name: sbc6713_top.bit
Open
Files of type: All Design Files
Cancel
// *** BATCH CMD : setMode -pff
// *** BATCH CMD : setMode -pff
// *** BATCH CMD : setMode -pff
// *** BATCH CMD : setSubmode -pffparallel
// *** BATCH CMD : setAttribute -configdevice -attr name -value "PFFConfigDevice"
// *** BATCH CMD : setAttribute -configdevice -attr size -value "0"
// *** BATCH CMD : addCollection -name "FILE_NAME"
// *** BATCH CMD : setAttribute -collection -attr dir -value "UP"
// *** BATCH CMD : addDesign -version 0 -name "0000"
// *** BATCH CMD : addDeviceChain -index 0
// *** BATCH CMD : setCurrentDesign -version 0
// *** BATCH CMD : setAttribute -design -attr name -value "0"
For Help, press F1
Prom Formatter

Figure 40. Select the BIT file

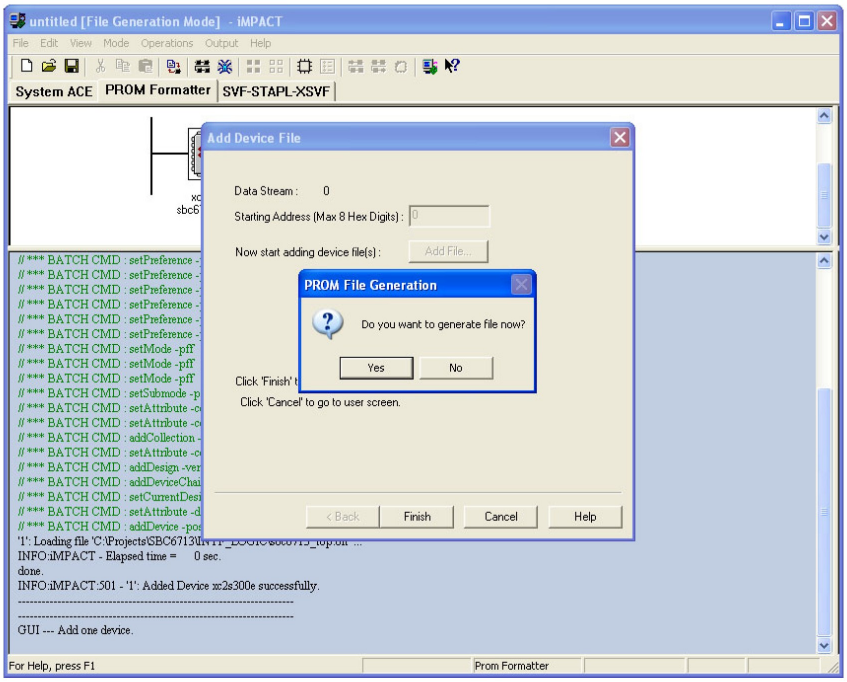


Figure 41. PROM File Generation.

Note: Please do the visual inspection of Spartan IIe(i.e. either 300E or 600E), and load the logic accordingly.

FlashBurn Utility

To load the new .EXO file into Spartan IIe (300K-600K), use the *FlashBurn.exe* utility included in the Pismo Toolset. Within the program, read in the EXO file you wish to use from the host disk, and program the Flash (AM29LV320DT). Reset the card in order to load the current logic in Spartan IIe. Below is the picture of *FlashBurn* utility.

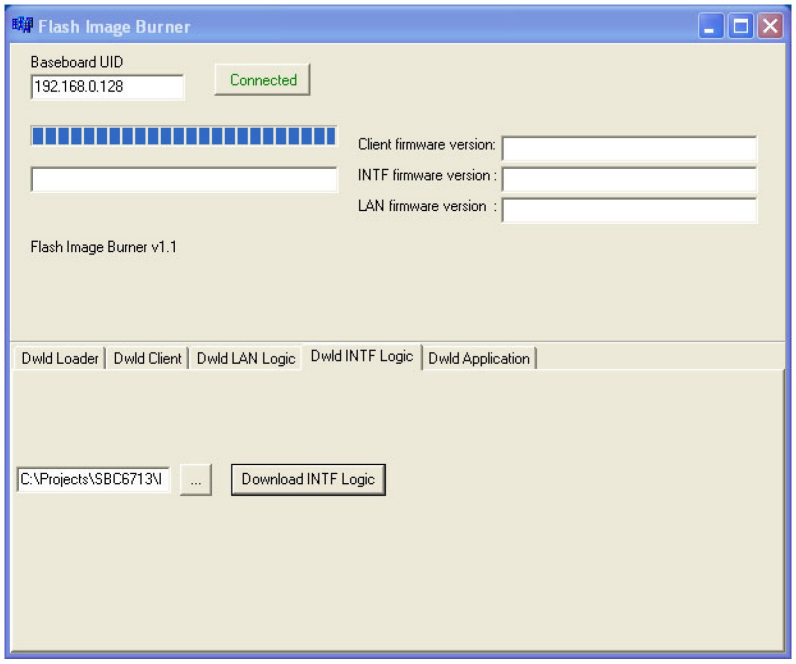


Figure 42. Flash Image Burner

In order to burn logic, the user needs to have an IP address dedicated to SBC6713e. As the user click on download, internally DM642 moves the data from SDRAM to the Flash to its specified sector. Then after, user needs to provide a board reset to load the current logic into FPGA (Spartan IIe). For further information about FlashBurn utility and its libraries, please refer to Pismo Toolset.

Debugging Custom Logic

It is inevitable that the logic will require some debugging and it is best to have a strategy for debug before you actually use the hardware. Debugging on actual hardware is difficult because you have poor visibility into the FPGA internals. There are several techniques that have worked for us on projects: Xilinx ChipScope, built-in test modes, and judicious use of testpoints. SBC6713e has three testpoints, in the logic, these are unused and may be connected to signals you want to monitor with a logic analyzer.

Built-in Test Modes

Another good way to debug your design is to have built-in test modes in the logic. If you plan ahead for test, then you can more quickly validate your design later and spot problems. When you finish the design, if the test generators and checkers can be left in the design, they are there later as production debug or test.

In many designs, test pattern or data generators are invaluable since they provide known data into the FPGA so that the output is known. If the data source is analog in the real design, substituting perfect data is nice because it helps spot

problems that may be hidden in the noise. The test pattern may be an easily recognized stream, like incrementing numbers, that are easy to check in the logic or on the test equipment. Also, its easier to test the extreme cases of the design that may be difficult to reproduce with real signals.

Also, data checkers in the logic sprinkled through the data chain help to spot the source of problems. If you have a missing timing constraint or a clock domain issue, these can be hard to catch since they may be rare. A data checker gives you a way to look for bad data and then trigger ChipScope or the logic analyzer. In many cases, rare errors are impossible to catch without this sort of data checker.

This technique has save a lot of time for big designs.

Xilinx ChipScope

Xilinx offers an excellent tool for debugging FPGA designs call ChipScope. This tool works over the FPGA JTAG port using any of the standard Xilinx JTAG cables. Software on the PC connects to a ChipScope logic core that you embed in your logic. This is an optional tool from Xilinx and is not included in the standard ISE software. For its cost of under $1000, we have found it well worth the money.

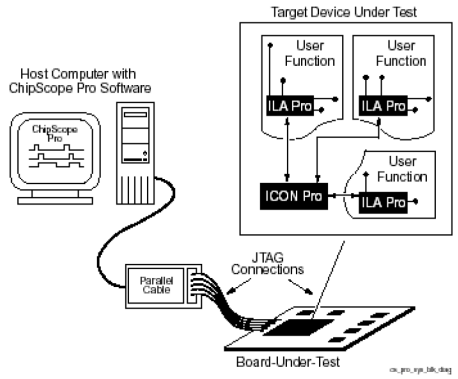


Figure 43. Chip Scope Big Picture.

The ChipScope core allows you to monitor internal FPGA signals that you have connected using triggers based on a set of trigger signals you have attached. A master clock connection is required for the core as well.

In order to debug custom logic using ChipScope, the user might need to burn the logic several times, so we have provided a JTAG (JP25) connector which is directly connected to Spartan IIe(300k-600K), it helps the user to load the logic as many times as needed.

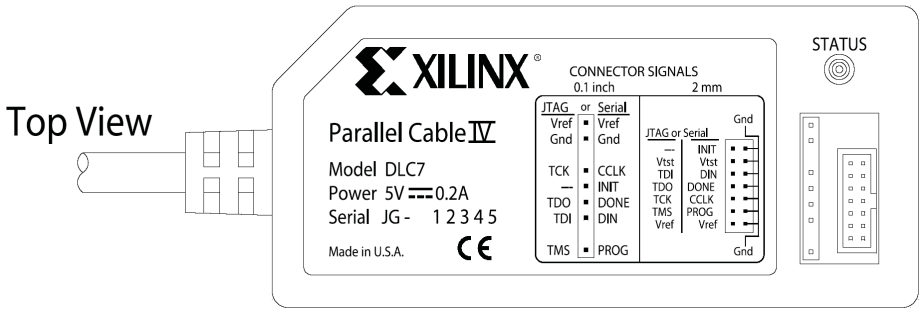


Figure 44. Xilinx Parallel Cable IV (Top View) (Courtesy of Xilinx)

Figure 45. High Performance Ribbon Cable (Courtesy Of Xilinx).

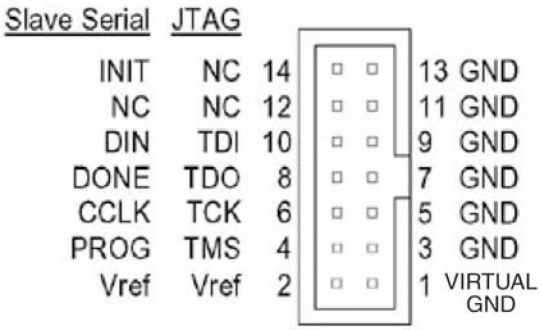

Slave Serial JTAG
INIT NC 14 13 GND
NC NC 12 11 GND
DIN TDI 10 9 GND
DONE TDO 8 7 GND
CCLK TCK 6 5 GND
PROG TMS 4 3 GND
Vref Vref 2 1 VIRTUAL GND

Figure 46. Xilinx Cable Interface Connector (Courtesy Of Xilinx).

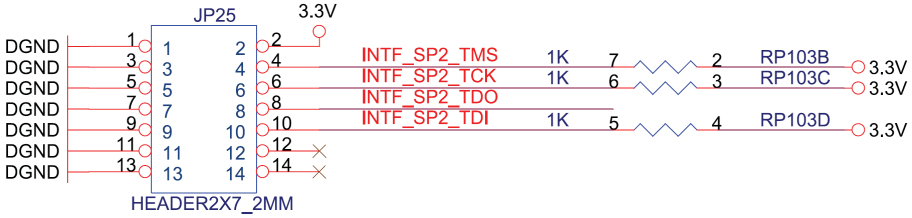


Figure 47. SBC6713e - JP25, Spartan IIe(300K-600K) JTAG Connector

Note: After loading the logic through JTAG, if you power cycle the card or if you hit the board reset, then the Spartan IIe is loaded with the logic you had in Flash ROM.

CAUTION: **The user SHOULD make sure that Xilinx Parallel Cable IV Target Interface Connector signal assignments matches with the Innovative header (JP25) on SBC6713e for Spartan IIe. If by mistake, the user connects the Xilinx cable incorrectly, there is a very good chance of damaging the SBC6713e card and Xilinx POD.**

Declaration Of ChipScope Core in VHDL

The ChipScope core is simple to use. Just connect up the signal for observation to the data ports, the trigger signals to trig and the clock. The number of ports and triggers is defined when you create the debug core in the ChipScope tool. Here's a core we used in debug shown below.

```
  --  ICON core signal declarations
  signal control0       : std_logic_vector(35 downto 0);

 ------------------------------------------------------------------
  --  ILA core component declaration

    component ila
    port
    (
      control     : in    std_logic_vector(35 downto 0);
      clk         : in    std_logic;
      data        : in    std_logic_vector(55 downto 0);
      trig0       : in    std_logic_vector(2 downto 0)
    );
  end component;

-----------------------------------------------------------------
  --  ILA core signal declarations
  signal control    : std_logic_vector(35 downto 0);
  signal clk        : std_logic;
  signal data       : std_logic_vector(55 downto 0);
  signal trig0      : std_logic_vector(2 downto 0);
```

Figure 48. Chip Scope Core Declarations.

Here is its instantiation during a debug session.

```
------------------------------CHIP SCOPE SIGNALS----------------------
-----------------------------------------------------------------------
 --
 --  ICON core instance
 --
-----------------------------------------------------------------------
  i_icon : icon
    port map
    (
      control0    => control0
    );

-----------------------------------------------------------------------
  --
  --  ILA core instance
  --
-----------------------------------------------------------------------
  i_ila : ila
```

```
    port map
    (
      control   => control,
      clk       => clk,
      data      => data,
      trig0     => trig0
    );

trig0(0)<= ce1_n_q;
--trig0(1)<= fpdp_rx_fifo_rd;
trig0(1)<= wdt_Wr;
trig0(2)<= ce2_n;

clk <= e_clk;
data(0) <= fpdp_tx_config(0);
data(2 downto 1) <= fpdp_tx_config(2 downto 1);
data(3)<=  (not fpdp_tx_fifo_int);
data(4)<=(not fpdp_rx_fifo_int);
data(5) <= fpdp_wrfifo_control_wr;
data(6) <= fpdp_rdfifo_control_wr;
data(7) <= dsp_data_oe_n(0);
data(8) <= fpdp_rx_fifo_rd;
data(12 downto 9) <= dsp_int_s;
data(13) <= not fpdp_tx_config(0);    --fpdp_reset
data(14) <=fpdp_tx_fifo_wr ;
--data(46 downto 15) <= fpdp_rx_do;
--data(55 downto 47) <= fpdp_rfifo_count;
data(27 downto 15) <= (others =>'0');
data(28) <= dsp_dll_locked;
data(29) <= reset_n;
data(30) <= reset;
data(31) <= wdt_count(20);
data(32) <= wdt_en;
data(33) <= '1' when ( (wdt_reset = '1' and wdt_en = '1') or dsp_force_reset = '1') else '0';
data(34) <= wdt_Wr;
data(55 downto 35) <= wdt_count;

control <= control0;
```

Figure 49. Chip Scope Core Instantiation and Use.

Once the core is in the design, you can then trigger on different conditions just as you would use a logic analyzer. If connect up all the signals in the problem area, only one compilation is needed to get the core into the design for debug.

Once you get it all working, you have a logic analyzer inside the FPGA. Here's sample view.

Figure 50. Chip Scope Analyzer Screen Shot.

For more information, see the Xilinx Website.

Creating a Binary File

When using single-board computers, such as the baseboard32 or baseboard6711, it is possible to embed application code into Flash ROM so that the application begins executing immediately after power-up. In order to accomplish this, you must first create a Code Composer project, build it, and then debug/test it by downloading it with the terminal emulator and Code Composer.

The standard Code Composer build process will generate a .out file, which is suitable for use within the terminal emulator or Code Composer, but is unsuitable for placement in Flash ROM. Once you have a viable .out file, it must be converted into a binary file before it can be burned into the target DSP card's Flash ROM. The process of converting an .out file into a binary file (.bin) is target specific.

baseboard31, baseboard32 and baseboard54 targets

To briefly describe this conversion process, the .out file from the Code Composer project will be converted to a .a0 file using a hex converter and a `.cmd` file. The `.a0` file is a readable ASCII version of the .out file. Then this intermediate .a0 file is converted to a `.bin` file using a hex to bin converter (`hex2bin.exe`) provided in your release. To accomplish this conversion quickly and easily, Innovative Integration has provided a template (`application.mk`) that can be modified to convert your file. This template is located in the target DSP card's root directory.

To convert your .out file into a .bin file to be burned to the Flash ROM, proceed as follows:

1. To start off, you must have a viable .out file (ie. yourfile.out).

2. Next you will have to create a `.mk` file for the out to bin conversion, using the template provided `application.mk`.

   - Copy the template, `application.mk` file (in `C:\<I.I. Target Board>`) to the directory where `yourfile.out` resides and rename it to `yourfile.mk`.

   - Open this new file with a text editing program like Notepad.

3. Edit the new .mk file as follows:

   - Change the OUTPUT and OUTPUT_BASE to the desired output name for the bin file. For example, to convert yourfile.out to yourfile.bin, edit this template file in the following manner:

     OUTPUT = yourfile.out

     OUTPUT_BASE = yourfile.

   - You must also be sure that the application.cmd and the hex2bin files are found by the new convertyourfile. To do this:

1) Change the HEX_ARGS to point the where the application.cmd file is (ie. if you have copied this file to the ..examples\target directory, then you must change HEX_ARGS = ..\..\application.cmd.

2) Similarly, you must point to the location of the hex2bin file in the HEX2BIN variable.

- Then save the changes and close the editing program.

  Note: It is always a good idea to remove or rename any old .bin file of the same name to ensure a new .bin file is created.

4. Open a DOS prompt and CD to the directory where the yourfile.out and the Convertyourfile.mk files reside. Type: **nmake -f yourfile.mk** and press <Enter>. This will create the yourfile.bin in the same directory.

baseboard67 and baseboard6711 Targets

Run the supplied applet PromImage.exe. Browse to your application .out file then click the Make button to generate the image file. The auto-generated image file will have the same name as the input .out file, but will be named with the .bin extension. This file is suitable for embedding into sector one of the flash ROM on the target board.

Note: baseboard67x must be equipped with version 1.2 or later of Talker in order to use these image files.

The Flash Burn Utility

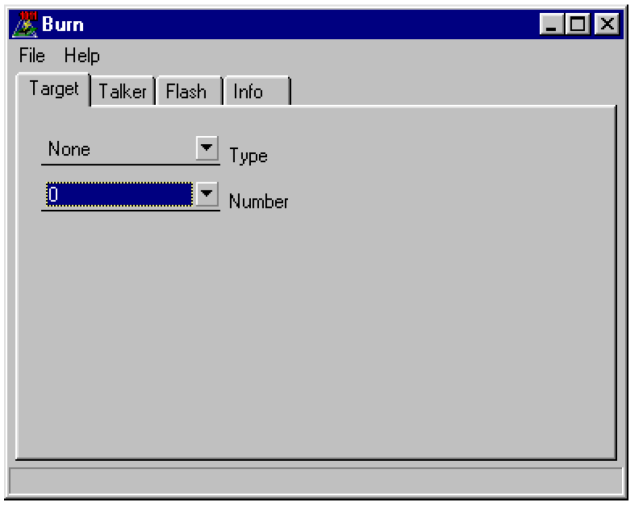
The Flash ROM programming utility (BURN.EXE) provides the capability to embed debugged application code within the flash ROM onboard Target DSPs which feature FLASH ROM. Currently, this includes each of the Innovative Integration single board computers (the baseboard31, baseboard32, baseboard54, baseboard62, baseboard67, and baseboard6711) products. Refer to the hardware manual for your DSP target to determine whether FLASH ROM is available on your system.

The utility supports both the 29F010 and 29F040 ROMs and provides special support for users with JTAG links between the host and the target in order to support programming.

The PROM utility contains only two menus, File and Help plus four tabbed page controls labeled Target, Talker, Flash, and Info. Currently, the Help menu is used to invoke help contents and the application About box, which indicates the revision of the application. The File menu provides access to the Options dialog which controls the timeout on JTAG loads and flash erase byte values (discussed later). In addition to the ability to convert Ascii hex text files into binary files, suitable for consumption by the Burn utility. The following is a brief description of the functions available under each of the primary tabs.

Target Page

The Target page contains two controls used to select the DSP type being used and its target number, as shown below.



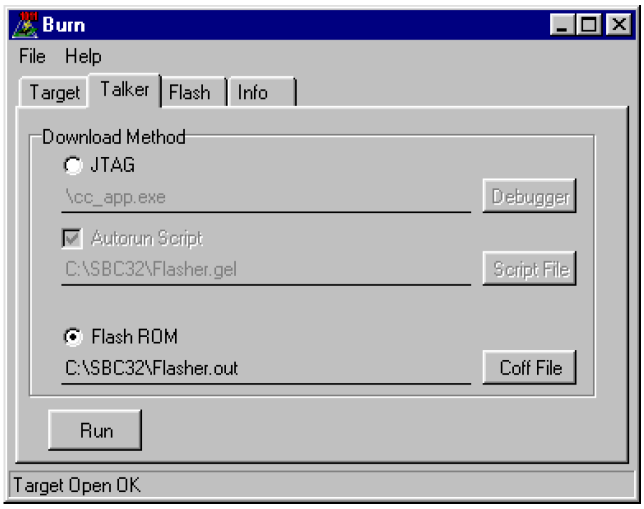
Click on the Number combo control and select the number of the DSP target you're using (usually 0). It is important to remember that for single-boards targets, zero corresponds to COM1 and target one corresponds to COM2, etc. Next, click on the Type combo control to display a list of supported targets, then select your DSP type.

After you have selected the DSP type, the applet will attempt to open the target device driver to establish communication with the target. The status of this attempt is displayed in the status bar at the bottom of the Burn applet window. If the driver is successfully opened (Target Open OK), click on the Talker tab to proceed. If the target driver does not successfully open, check the parameters being requested and the installation of your DSP. Also, close any other open applications (such as the terminal emulator.EXE, which could have previously opened the DSP device driver. Due to Windows restrictions, only a single application may open a com port device at a time.

Talker Page

To support all single-board DSP targets similarly, BURN must download a simple DSP support executable to the target before an embedded application may be burned into the Flash ROM. This executable is FLASHER.OUT, located in the root directory of the Zuma Toolset. This small, pre-written executable provides all of the Flash I/O access functions necessary to allow the BURN utility to interact with the target Flash ROM.

The Talker page contains a number of window controls used to specify the mechanism to use to deliver the FLASHER.OUT support executable into the target DSP. Two methods are currently supported, JTAG and downloading via the existing Talker in Flash ROM.



You must use the JTAG download method whenever the target DSP Flash ROM does not contain a viable (bootable) image of the Talker program. In the default factory condition, the target contains a bootable Talker, which is used by application programs through the DLL and device driver to download executable programs to the target DSP. If this Talker image has been erased or corrupted, you must use the JTAG download method.

If the Talker is still viable in Flash, then you may use either of the download method. The Talker method is the preferred method, especially in circumstances where a JTAG debugger is unavailable, such as when performing in-the-field software updates.

JTAG Download

To download using JTAG, click the JTAG radio button and then click the Debugger button to browse to the location of your JTAG debugger software (usually `c:\composer\cc_app.exe` if you're using Code Hammer or `c:\ti\cc\bin\cc_app.exe` if you're using Code Composer Studio) and click `<Open>`. Next, click on the Script File button to browse to the location of the Flasher.gel file located in the root directory of the board-specific Zuma libraries (ie. `c:\baseboard6711\Flasher.gel)` and click `<Open>`.

Under most circumstances, you should enable the Autorun Script check box as well. With this feature enabled, the specified Code Composer GEL script file is automatically executed as the debugger is started. A default script file has been provided and is located in the root directory of the Zuma Toolset installation. This GEL file automatically initializes the target DSP, downloads the `Flasher.out` support executable and launches it using the debugger Run Free command.

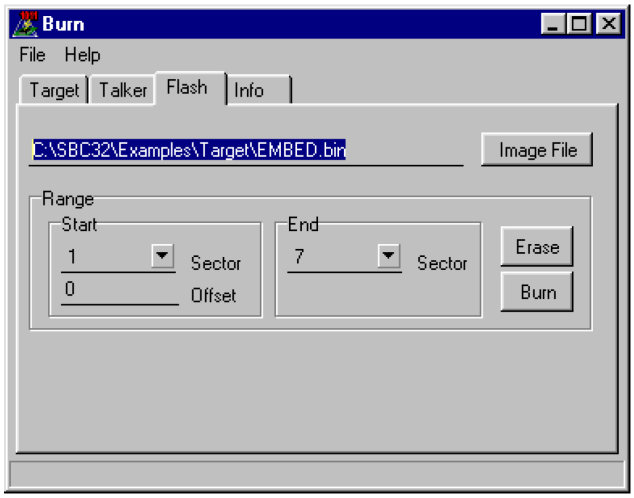
In rare circumstances, you may elect to disable the Autorun Script option. When this feature is disabled, BURN will automatically launch the debugger when you click the Run button, but you must manually load and run the `Flasher.out` executable. For example, this option is useful when using a debugger other than Code Hammer.

Flash ROM Download
To download using the Talker in flash ROM, click on the Flash ROM radio button and then click on the Coff File button to browse to the location of the `Flasher.out` file. This file is usually located in the root of the board-specific Zuma libraries directory (ie `c:\baseboard6711\Flasher.out`).

Downloading the Flash Support Code
Close Code Composer, if you had been using the debugger, click on `Debug|Run Free` before exiting. To download the flash support executable, `Flasher.out`, click on the Run button. The program will deliver the `Flasher.out` executable image into the Target using the selected method. When the support code has been download and is running, the status bar will display Flash Info read OK. If you have difficulty running the support executable, refer to the "Common Problems" section for troubleshooting suggestions.

Flash Page

When the support code has been downloaded and is running, the status bar will display "Flash Info read OK". Next, you may burn binary images into the Flash ROM of the target DSP. Click on the Flash tab to change to the Flash Burn page (seen below).



Click on the Image File button to select the binary image file that is to be burned into the DSP flash. Note that BURN is incapable of burning DSP COFF executables (.OUT files) directly. You must use the other tools in the Zuma Toolset (specifically CodeWright, TIDeps and the TI HEX Conversion Utility) to create a viable binary image before attempting to burn an application into Flash ROM. Example embed-able projects for each II single-board target are available on the Innovative web site at www.innovative-dsp.com.

Controlling Region of Erasure
Adjust the Start Sector and End Sector combo boxes within the Start and End group boxes to control the location within the target flash in which the image is to be placed. Note that this controls only the range of sectors erased prior to attempting to burn the application into Flash.

The number of sectors actually needed is determined by the size of the binary image file. The BURN applet performs a range check to insure that your image will actually fit into the erased sectors. You must be sure to erase a sufficient number of sectors to contain your entire application image.

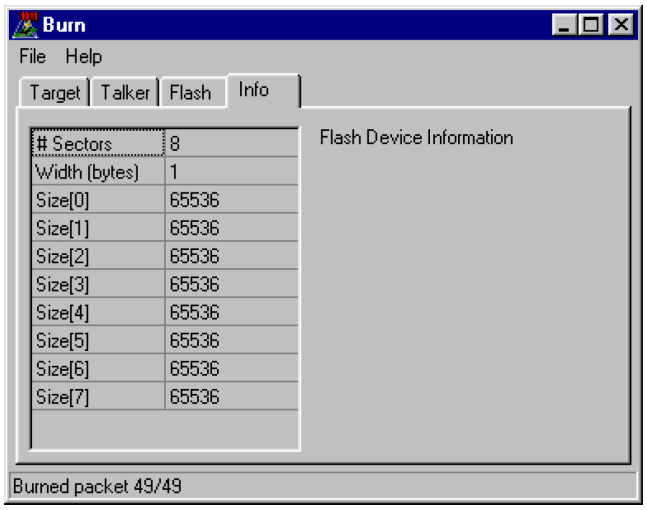
Re-burning the Talker Image
To burn or re-burn the factory Talker image, select 0 as both the starting and ending Sector. On all targets except C32-based targets, select 0 as the Offset. On C32 targets, the Offset should be set to 0x1000.

Warning:
Avoid burning application code other than the factory Talker image into sector zero. On some targets, notably the baseboard62, it may be impossible to initialize the JTAG debugger on a target board containing an invalid boot image. This situation can only be corrected by shipping the target DSP back to the factory for rework.

Burning a User-Written Application
To burn a user-written application, select 1 as the starting Sector and adjust the ending sector to the uppermost available sector, less any reserved sectors used as data storage within your DSP application program. The Offset field should be set to zero for all target board types.

Info Page

Detailed autoselect information read from the Flash ROM is available on this Info Tab. This information is utilized internally by the BURN applet. You need not enter information on this tab.



Example Burn Sequence

The following is a typical example of burning a binary file (`.bin`) into the target card's Flash ROM.

1. Run the Flash ROM programming utility (`BURN.EXE`) from the `Start Menu/Programs/<Target Board>/Burn`.

2. Click on the Target Tab.

   - Then click on the Number combo box and select the number of the DSP target you are using (usually 0).

   - Next, click on the Type combo box and select the target DSP type. (At the bottom of the display, "Target Open OK" should be displayed)

3. Click on the Talker Tab.

   - In this example, from the two download methods, the Flash ROM method is selected. Click on the `<Coff File>` button and browse to the root directory of the Zuma Toolset and find the Flasher.out file.

   - Then, click on the `<Run>` button. (At the bottom of the display, "Flash info read OK" should be displayed)

4. Click on the Flash Tab.

   - Then, click on the `<Image File>` button and browse to the binary file that is to be downloaded to the target DSP flash.

   - Next, adjust the Start Sector combo box to 1 and the End Sector combo box to 7.

   - Then, click on the `<Burn>` button to start the download. The downloading progress should be displayed. (At the bottom of the display, "Burn Completed" should be displayed)

5. Testing the burned .bin file.

baseboard32:

- To run the application from the target DSP, turn the power off, disconnect the JTAG & serial port, and turn the power back on. Alternately with the serial cable attached, you may click on the Boot Embedded Application speed button on the terminal emulator to launch the embedded application, rather than the embedded Talker.

baseboard6x:

- To run the application from the target DSP, turn the power off, install jumper (JP24) and turn the power back on. Alternately with the serial cable attached and JP24 installed, you may click on the Reset Dsp speed button on the terminal emulator to launch the embedded application, rather than the embedded Talker.

baseboard6711:

- To run the application from the target DSP, turn the power off, install jumper (JP11) and turn the power back on. Alternately with the USB cable attached and JP11 installed, you may click on the Reset Dsp speed button on the terminal emulator to launch the embedded application, rather than the embedded Talker.

Common Problems when Embedding Code

There are several problems frequently encountered when ROMing code. If you encounter one of the symptoms listed below, attempt the corrective action listed before calling Innovative technical support:

Table 15. Common Problems when Embedding Code in Flash ROM

Symptom	Possible Corrective Actions
Inadvertently overwritten the Talker in sector 0.	**Re-burn the Talker (per above).**
Code Hammer cannot initialize the target.	Verify that the I/O address specified in the Code Composer setup is correct and that the JTAG board is properly connected to the DSP board. Configure and run the Innovative JTAGDIAG.EXE utility to reset the debugger hardware. Code Composer Studio users should also run the XDS510 Reset Utility (using the same I/O address as entered into JTAGDIAG.EXE) to initialize the Studio Debugger. The target may be held in reset. Verify that the boards device driver is installed and operational. Code Composer versions prior to 4.01 are incapable of communicating with C67xx processors. Also, v4.01 is better able to recover from invalid images burned into sector 0 on baseboard6711 DSPs. Contact I.I. for upgrade information.
The embedded application will not boot.	The target is being held in reset. For single-board targets, disconnect serial port #1 to the host and reboot. If serial port #1 must be connected, insure that the DTR output from the PC is de-asserted (since it controls target reset). Further, insure that the baseboard's CTS serial input line is de-asserted since the Talker

	asserted during cold- boot. On the baseboard62, insure that the boot jumper JP24 is installed. You burned the application at an incorrect offset or starting at a sector other than 1. The application burn image was created improperly. Inspect the `.A0` file and insure that the target image has an appropriate boot record. Be sure to convert the `.A0` file into binary using the Options dialog for consumption by the Burn applet. On the baseboard62 and baseboard54 targets, be sure to locate the `.const` and cinit sections into the ROM so that they may be resurrected at boot time. The Target may still be held by the JTAG. Run the JTAG Diagnostic utility and then select the <Reset> button to release the target.
The application works properly from within Code Composer Studio but does not run (or is erratic) when embedded.	The `.bss` section of your application is not automatically initialized to zero. Modify your linker command file to zero-fill the .bss section. Your embedded application may be too large. Use the Coff Dump utility to inspect the size of the .OUT file to verify that it can fit into the seven residual 16 kByte sectors in the AMD 29F010 (64 kByte sectors in the 'F040).

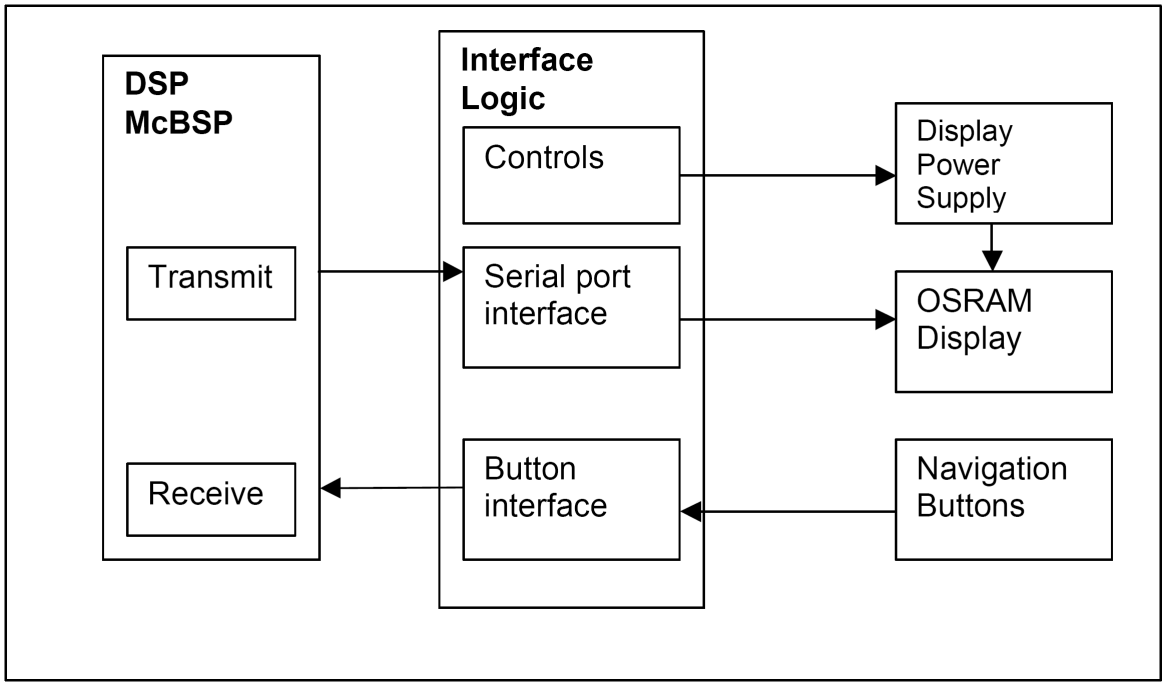
OLED Display Adapter

Description

This assembly provides an interface from a TI McBSP to an OSRAM Pictiva OLED graphic display using a Solomon SSD0323 display controller.

This adapter supports OSRAM displays OS128064 that are 128 by 64 pixels. An optional power supply on the adapter board may be used with displays assemblies that do not have a built-in display power supply.

This adapter is for use with a serial data interface and a navigation switch assembly. The display adapter translates serial data from the DSP McBSP port into serial data and commands for the display and adapter. Read back of the 5 buttons used for navigation includes debounced switch event data to the DSP via an serial word transmitted back to the DSP.



Button Interface

The display adapter supports a navigation button assembly that has five switches integrated into a single button assembly. The five buttons (UP, DOWN, RIGHT, LEFT, SELECT) are integrated into a joystick-like button assembly with a single keycap that swivels and depresses. The switch is ITT Cannon TPA.

The buttons are debounced by sampling the button position with a clock equal to the DSP McBSP clock divided by 2^13. For a 3 MHz clock, this is about a 2.7 mS sample period. The switch must be sampled 16 consecutive times at '1' or '0' to result in a debounced output of that value.

Once a debounced switch is sensed, any change in the switch value from its previous state results in a serial word being sent to the DSP McBSP serial port that reports all switch values currently and some status. This transmission is ONLY done if a change is sensed.

Bit	Function	Meaning
0	Left	'1' = left is depressed
1	Right	'1' = right is depressed
2	Down	'1' = down is depressed
3	Up	'1' = up is depressed
4	Select	'1' = select is depressed
7..5	"010"	-

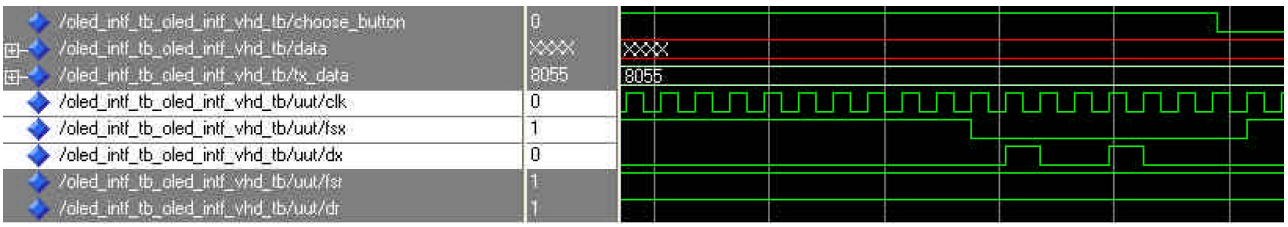
Note that it is possible for multiple switches to be simultaneously.

The McBSP receive port should be configured as

Frame sync = input, active low, 8 bits wide, frames the data byte

Data = 8 bit, clocked on falling edges of the clock, MSB received first

Clock = input, falling edges used for frame and data



This timing diagram shows the serial data transmission from the logic to the DSP with the UP button true.

Display Interface

The display controller, Solomon SSD0323, has a synchronous serial data interface that is adapted to use with the TI DSP serial port (McBSP). The McBSP transmit port is used as both an adapter and display command and control.

The McBSP should be configured to transmit a 16-bit word with a frame active during the data transmission. The McBSP should continuously source the serial clock at 3.33 MHz maximum. The frame is sent by the McBSP and is active low.

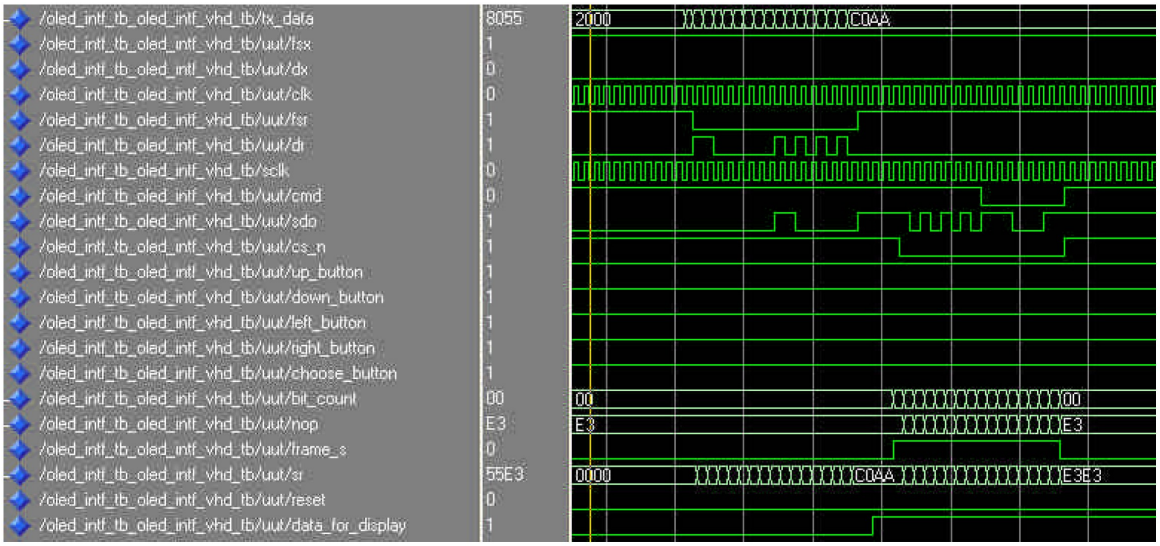
The adapter receives the data and parses the data word. If the data is addressed to the display (bit 15 = '1'), the data is sent to the display. If it is a command to the display, then the lower byte is sent to the display as an 8-bit transmission. If it is data for the display, then the data is sent with a NOP command (0xE3) automatically appended to it as required by the controller.

Serial Word Format

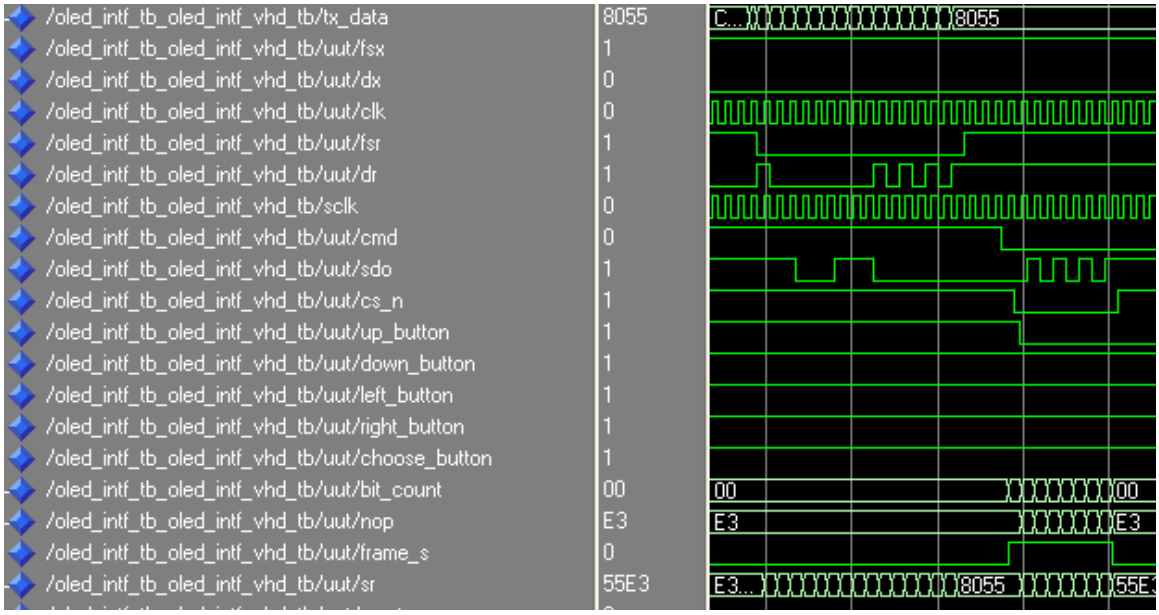
The serial data word format is as follows.

Bit	Function	Meaning
7..0	Display command or data word	See SSD0323 data sheet for command set.
10..8	Not used	-
11	Display power	'1' = on, '0' = off (default)
12	Display reset	'1' = reset (default)
13	Adapter reset	'1' = reset, '0' = not in reset(default)
14	Display CMD	'1' = bits 7..0 are display data, '0' = bits 7..0 are display command
15	Device select	'1' = this data word is for the adapter; '0' = this data word is for the display

An example transaction is shown here. The DSP has sent a data word of 0xC0AA, representing that the display should receive a data byte of 0xAA. The data is then immediately sent to the display as a 16-bit transmission of data = 0xAA followed by a NOP command.



For commands sent to the display, the adapter logic only sends an 8-bit word to the display. The following transaction shows the DSP sending a command to the display of 0x8055. A command of 0x55 is subsequently sent to the DSP.



Data/Command Pacing

There is NO pacing method used on the serial port. Transmissions to the adapter therefore must not be done any faster than every 11 µS for data words, or 8 µS for command words to the display. Adapter commands require 1 µS between serial receptions. These timings assume a 3.33 MHz serial clock (max recommended rate) so adjust them as necessary for other serial clock rates.

Required steps to use the display

1. Reset the adapter board. This is done by writing a serial word with bit 13 true. This bit is NOT sticky; you need write it only once true.

2. Turn on the display power. Allow 100 mS for stabilization.

3. Reset the display.

4. Begin display initialization as described in the SSD0323 manual.

Appendices

Connector pinouts

JP4, JP8 - OMNIBUS I/O Connectors

Connector Types:	AMP .05 Subminiature D male, AMP 173280-3
Number of Connections:	50
Mating Connector:	AMP 173279-3

JP5- OMNIBUS I/O Connectors

Connector Types:	MDR connector, 3M N102A0-52E2VC
Number of Connections:	100
Mating Connector:	3M 101A0-4CZ3JL

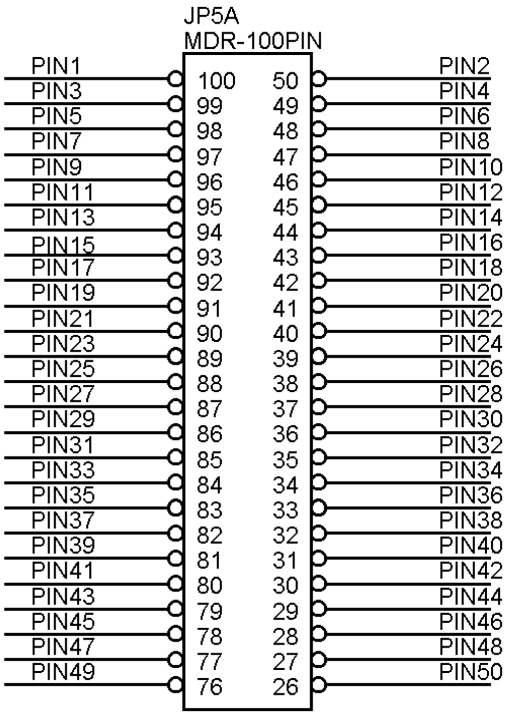
The following table shows the interconnections between the JP4 (OMNIBUS slot 0) and JP5 (OMNIBUS IO connector).

JP4, Module 0 Pin	JP5 Pin Numbers
1	100
2	50
3	99
4	49
5	98
6	48

JP4, Module 0 Pin	JP5 Pin Numbers
7	97
8	47
9	96
10	46
11	95
12	45

JP4, Module 0 Pin	JP5 Pin Numbers
13	94
14	44
15	93
16	43
17	92
18	42
19	91
20	41
21	90
22	40
23	89
24	39
25	88
26	38
27	87
28	37
29	86
30	36
31	85

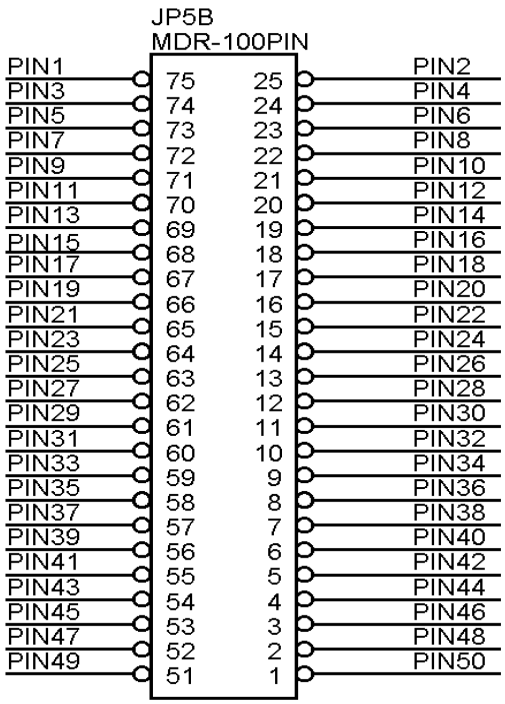
JP4, Module 0 Pin	JP5 Pin Numbers
32	35
33	84
34	34
35	83
36	33
37	82
38	32
39	81
40	31
41	80
42	30
43	79
44	29
45	78
46	28
47	77
48	27
49	76
50	26

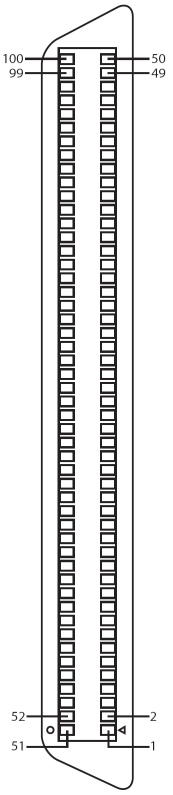


The following table shows the interconnections between the JP8 (OMNIBUS slot1) and JP5 (OMNIBUS IO connector).

JP8, Module 1 Pin	JP5 Pin Numbers
1	75
2	25
3	74
4	24
5	73
6	23
7	72
8	22
9	71
10	21
11	70
12	20
13	69
14	19
15	68
16	18
17	67
18	17
19	66
20	16
21	65
22	15
23	64
24	14
25	63

JP8, Module 1 Pin	JP5 Pin Numbers
26	13
27	62
28	12
29	61
30	11
31	60
32	10
33	59
34	9
35	58
36	8
37	57
38	7
39	56
40	6
41	55
42	5
43	54
44	4
45	53
46	3
47	52
48	2
49	51
50	1


JP5B
MDR-100PIN
PIN1
PIN3
PIN5
PIN7
PIN9
PIN11
PIN13
PIN15
PIN17
PIN19
PIN21
PIN23
PIN25
PIN27
PIN29
PIN31
PIN33
PIN35
PIN37
PIN39
PIN41
PIN43
PIN45
PIN47
PIN49
75
74
73
72
71
70
69
68
67
66
65
64
63
62
61
60
59
58
57
56
55
54
53
52
51
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
PIN2
PIN4
PIN6
PIN8
PIN10
PIN12
PIN14
PIN16
PIN18
PIN20
PIN22
PIN24
PIN26
PIN28
PIN30
PIN32
PIN34
PIN36
PIN38
PIN40
PIN42
PIN44
PIN46
PIN48
PIN50


100
99
50
49
52
51
2
1

JP6, 7, 9, 10 – OMNIBUS Bus Connectors

Connector Types:	AMP .05 Subminiature D male
Number of Connections:	50
Mating Connector:	AMP 173279-3

The following table gives the pin numbers and functions for the JP6 (OMNIBUS slot 0) and JP9 (OMNIBUS slot 1) connectors. The functions for JP9 are identical to those of JP6, except where noted.

Table 16. OMNIBUS Bus Connectors

Pin Number	JP6 Function	JP9 Function	Direction (from SBC6713e)
1, 19	Digital +5V		O, power
2, 20	Digital ground		O, power
3-18	Data bus 0-15		I/O
21, 43, 40, 45, 39, 26, 27	Address bus 2-8		O
28	Reset (active low)		O
29	External interrupt 0	External interrupt 2	I
30	Bus ready (active low)		I (open-collector)
31	Processor ECLK/2 (37.5 MHz)		O
32	DSP timer channel 0		O
33	R/W*		O
34	DDS timebase		O
35-38	IOMOD0-3 decoded selects (active low)	IOMOD4-7 decoded selects (active low)	O
25	Analog 15V (OMNIBUS 12V)		O, power
23	Analog +15V (OMNIBUS +12V)		O, power
41,42	Analog ground		O, power
22,24	Analog -15V		O, power
44,46	Analog +15V		O, power
47,49	Analog +5V		O, power
48,50	Analog -5V		O, power

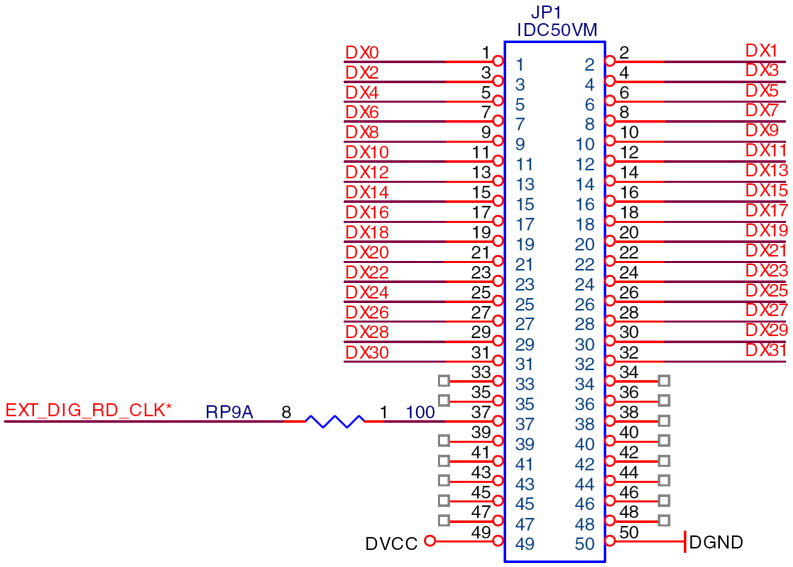
The following table gives the pin numbers and functions for the JP7 (OMNIBUS slot 0) and JP10 (OMNIBUS slot 1) connectors.

Table 17. I/O Module Bus Connectors

Pin Number	JP7 Function	JP10 Function	Direction (from SBC6713e)
1, 3-6	Address bus 9-13		O
2, 19, 20, 49, 50	Digital ground		O, power
7, 9, 11-13, 15, 17	Reserved	Reserved	NA
8, 10	Digital +3.3V		Power
14	Module trigger 0		O
16	Module trigger 1		O
18	Module trigger 2		O
21	Processor timer channel 1		O
22	Module trigger in	External trigger 1	O
23,25	Analog +15V (OMNIBUS +12V)		O, power
24	CLKS0	CLKS1	I
26	CLKR0	CLKR1	I/O
27	FSR0	FSR1	I/O
28	CLKX0	CLKX1	I/O
29	External interrupt 1	External interrupt 3	I
30	DR0	DR1	I
31	FSX0	FSX1	I/O
32	DX0	DX1	O
33-48	Data bus 16-31		I/O

JP1 – Digital I/O Connector

Connector Types:	0.1” double row shrouded header, center bump polarized, Thomas and Betts 609-5027
Number of Connections:	50
Mating Connector:	AMP 1-746285-0



The following table gives the pin numbers and functions for the JP1 connector.

Table 18. Digital I/O Connector

Pin Number	JP1 Function	Direction (from SBC6713e)
1-32	Digital I/O bit 0..31	I/O
33-36	Not used	-
37	External Digital Readback Clock(latch falling edge)	I
38-48	Not used	-
49	DVCC (digital +5 V)	Power
50	DGND (digital ground)	Power

JH1 – FPDP Transmit Port Connector

Connector Types:	P50E-080P1-S1-TG
Number of Connections:	80
Mating Connector:	P25E-080S-TGF (Manuf.3M)

The following table gives the pin numbers and functions for the JH1 connector.

Pin Number	JH1 Function	Direction (from SBC6713e)
1,3,4,5,6,8,10, 12,14,16,18,20 ,22,24,26,28, 30,32,35,38,41 ,44,47,50,53, 56,59,62,65,68 ,71,74,77,80	Digital Ground	Power
2	Tx Strobe	O
7	Tx NRDY N	O
9	Tx DIR N	O
13	Tx Suspend N	O
17	Tx PIO2	O
19	Tx PIO1	O
25	Tx PStrobe P	O
27	Tx PStrobe N	O
29	Tx Sync N	O
31	Tx DValid N	O
33, 34, 36, 37	TxD31, TxD30, TxD29, TxD28	O
39, 40, 42, 43	TxD27, TxD26, TxD25, TxD24	O
45, 46, 48, 49	TxD23, TxD22, TxD21, TxD20	O
51, 52, 54, 55	TxD19, TxD18, TxD17, TxD16	O
57, 58, 60, 61	TxD15, TxD14, TxD13, TxD12	O
63, 64, 66, 67	TxD11, TxD10 , TxD9 , TxD8	O
69, 70, 72, 73	TxD7 , TxD6 , TxD5 , TxD4	O
75, 76, 78, 79	TxD3 , Tx2D , TxD1,	O

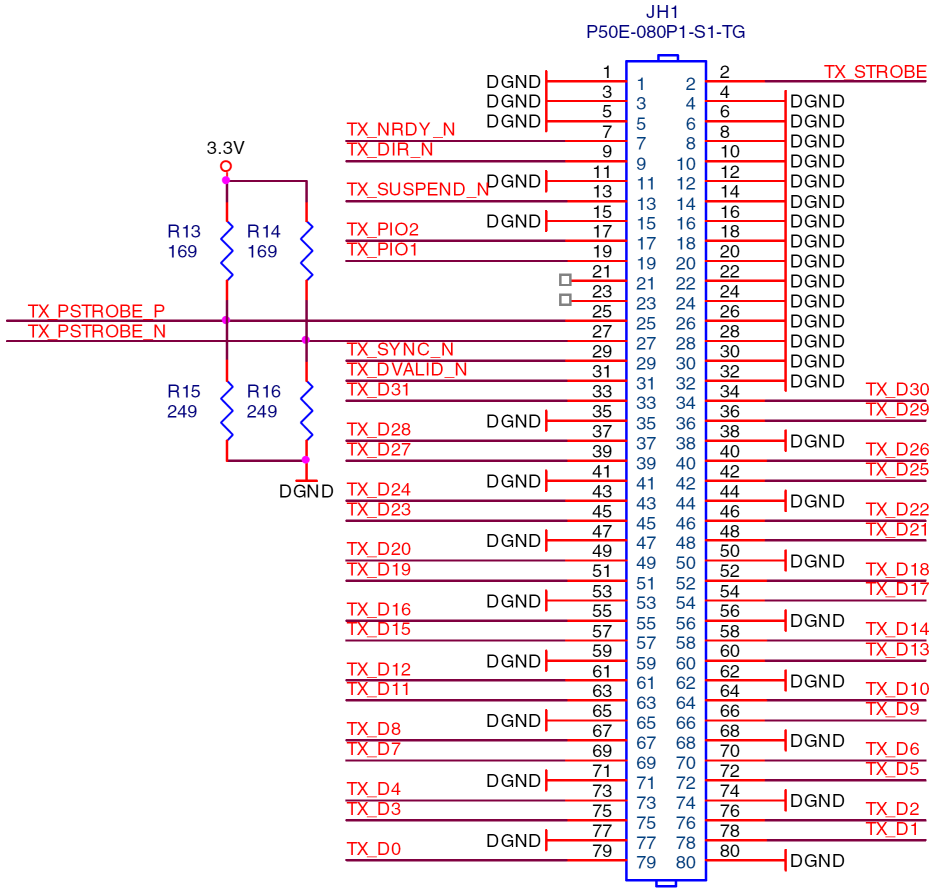


Figure 51. FPDP JH1 Tx Port Connector

JH2 – FPDP Receive Port Connector

Connector Types:	P50E-080P1-S1-TG
Number of Connections:	80
Mating Connector:	P25E-080S-TGF (Manuf.3M)

The following table gives the pin numbers and functions for the JH2 connector.

Pin Number	JH2 Function	Direction (from SBC6713e)
1,3,4,5,6,8,10, 12,14,16,18,20 ,22,24,26,28, 30,32,35,38,41 ,44,47,50,53, 56,59,62,65,68 ,71,74,77,80	Digital Ground	Power
2	Rx Strobe	I
7	Rx NRDY N	I
9	Rx DIR N	I
13	Rx Suspend N	I
17	Rx PIO2	I
19	Rx PIO1	I
25	Rx PStrobe P	I
27	Rx PStrobe N	I
29	Rx Sync N	I
31	Rx DValid N	I
33, 34, 36, 37	RxD31, RxD30, RxD29, RxD28	I
39, 40, 42, 43	RxD27, RxD26, RxD25, RxD24	I
45, 46, 48, 49	RxD23, RxD22, RxD21, RxD20	I
51, 52, 54, 55	RxD19, RxD18, RxD17, RxD16	I
57, 58, 60, 61	RxD15, RxD14, TxD13, RxD12	I
63, 64, 66, 67	RxD11, RxD10, RxD9 , RxD8	I
69, 70, 72, 73	RxD7 , RxD6 , RxD5 , RxD4	I
75, 76, 78, 79	RxD3 , Rx2D , RxD1,	I

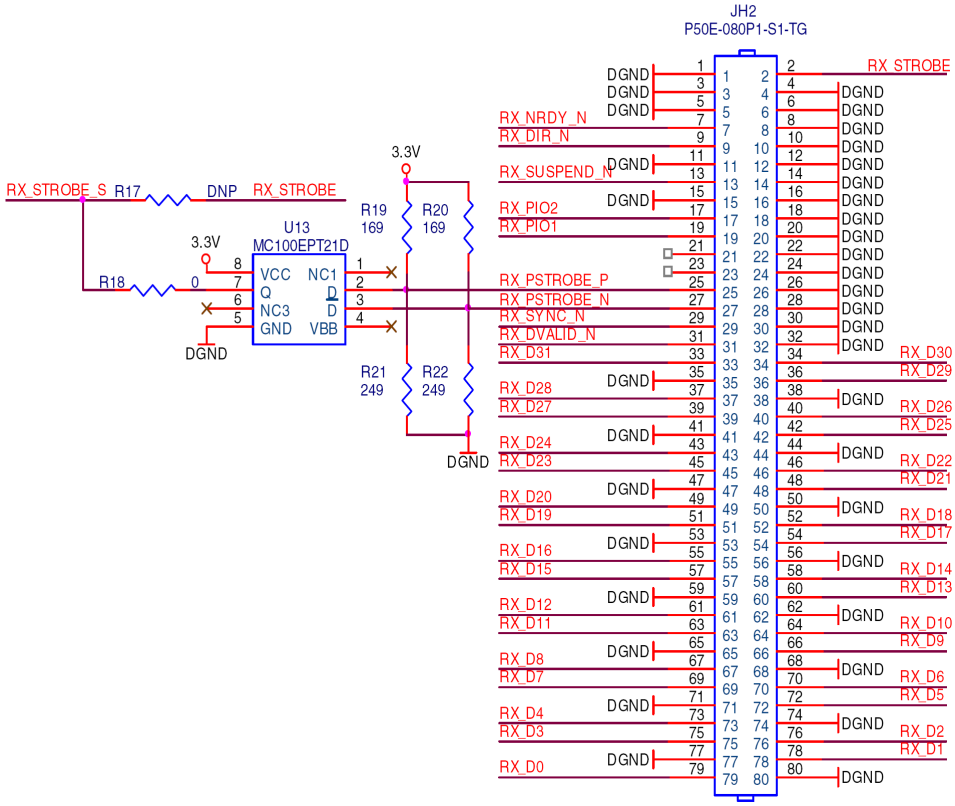


Figure 52. FPDP JH2 Rx Port Connector

JP2 - SyncLink/ClkLink

Connector Types:	0.1” double-row shrouded header
Number of Connections:	14
Mating Connector:	AMP

The SyncLink connector allows SBC6713e to synchronize to external hardware or other Innovative cards.

The following table gives the pin numbers and functions for the JP2 connector.

Table 19. SyncLink Connector

Pin Number	JP2 Function	Direction (from SBC6713e)
1	Clocklink Out +	O
2	Clocklink Out -	O
3	Clocklink In +	I
4	Clocklink In -	I
5	Synclink bus pin 2	I/O
6	Digital ground	Power
7	Synclink bus pin 1	I/O
8	Digital ground	Power
9	Synclink bus pin 0	I/O
10	Digital ground	Power
11	Synclink bus pin 3	
12	Synclink bus pin 5	
13	Synclink bus pin 4	
14	NC	-

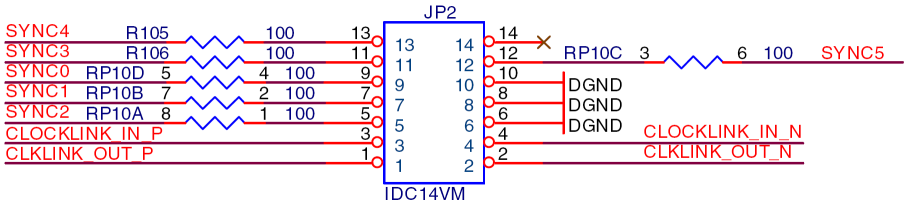


Figure 53. JP2 SyncLink Connector Pin Orientation

JP13, JP15 – Processor Serial Port Connectors

Connector Types:	2 mm double row header
Number of Connections:	10
Mating Connector:	Samtec SQT style (for board-board applications)

The following table gives the pin numbers and functions for the JP15 (McBSP 0) and JP13 (McBSP 1) connectors. Pin functions of JP15 are identical to those of JP13 except where noted.

Table 20. Processor Serial Port Connector

Pin Number	JP15 Function	JP13 Function	Direction (from SBC6713e)
1	CLKS0	CLKS1	I
2	FSR0	FSR1	I/O
3	CLKR0	CLKR1	I/O
4	FSX0	FSX1	I/O
5	CLKX0	CLKX1	I/O
6	Digital 3.3V	Digital 3.3V	Power
7	DR0	DR1	I
8	Digital 5V	Digital 5V	Power
9	DX0	DX1	O
10	Digital Ground	Digital Ground	Power

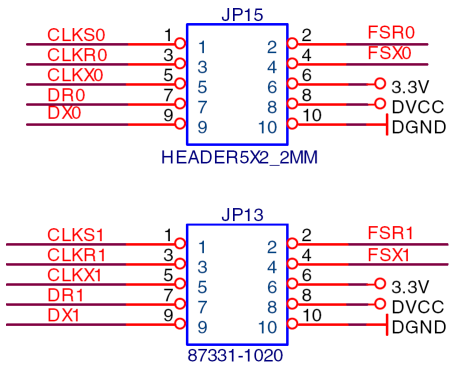


Figure 54. JP13, JP15 DSP Serial Port Connector

JP23 – JTAG Debugger Connector (Rev C)

Connector Types:	Shrouded header, pin 6 removed for key
Number of Connections:	14
Mating Connector:	AMP 746285-2

The following table gives the pin numbers and functions for the JP8 connector.

Rev B : JP8 ONLY connects to the 6713 DSP JTAG port.

RevC and onward: The JTAG path for the DSP and DM640 have been combined into a single scan path with the 6713 first followed by the DM640.

Table 21. JTAG Debugger Connector

Pin Number	JP8 Function	Direction (from SBC6713e)
1	TMS	I
2	TRST*	I
3	TDI	I
5	Digital +3V	Power
7	TDO	O
9,11	TCK	I
13	EMU0	I/O
14	EMU1	I/O
4, 6, 8, 10, 12	Digital ground	Power

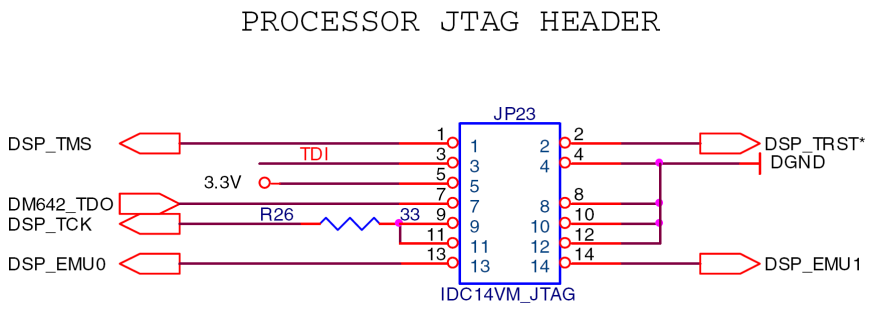


Figure 55. JP23 JTAG Debugger Connector

JP22 – Power Input Connector

Connector Types:	6 pin locking power connector (Molex 43045-0602)
Number of Connections:	6
Mating Connector:	Molex 43025-0600 and contacts

The following table gives the pin numbers and functions for the JP22 connector.

Table 22. Power Input Connector

Pin Number	JP22 Function	Direction (from SBC6713e)
1	Digital +5V	Power
2,4	Digital ground	Power
3	Analog +15V	Power
5	Analog 15V	Power
6	MOD_PWR power control	O

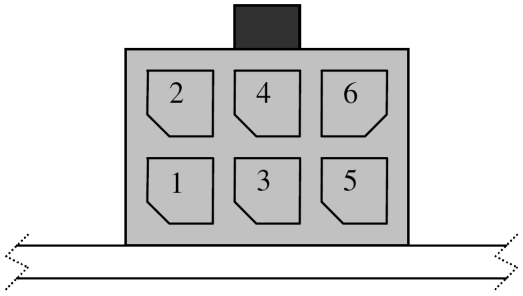


Figure 56. Power Connector Pin Positions (side view, from front of connector, showing connector keying and locking tab along with printed circuit board position)

Note: Matting connector may be numbered differently.

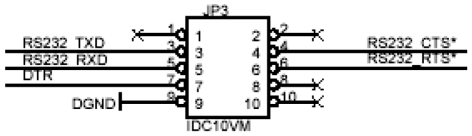
JP3 – Asynchronous Serial Port Connector

Connector Types:	Shrouded header
Number of Connections:	10
Mating Connector:	AMP 746285-1

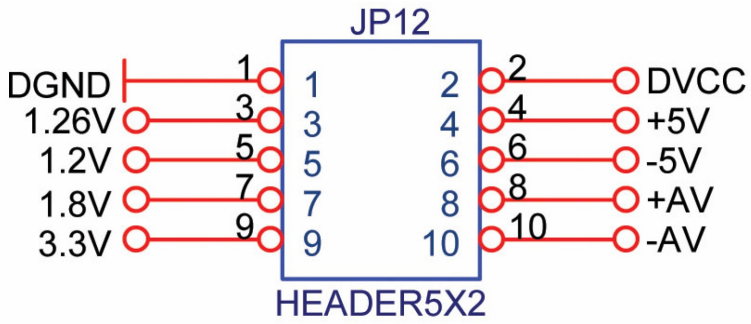
The following table gives the pin numbers and functions for the JP3 connector.

Table 23. Asynchronous Serial Port Connector

Pin Number	Function	Direction (from SBC6713e)
1, 2, 8, 10	Reserved	NA
3	RS232 Transmit Data	O
4	RS232 CTS	I
5	RS232 Receive Data	I
6	RS232 RTS	O
7	RS232 DTR	I
9	Digital ground	Power



JP12 - Power Test Connector (Rev C)



JP25 - Xilinx JTAG Connector (i.e. XC2Sx00E-FG456) (Rev C)

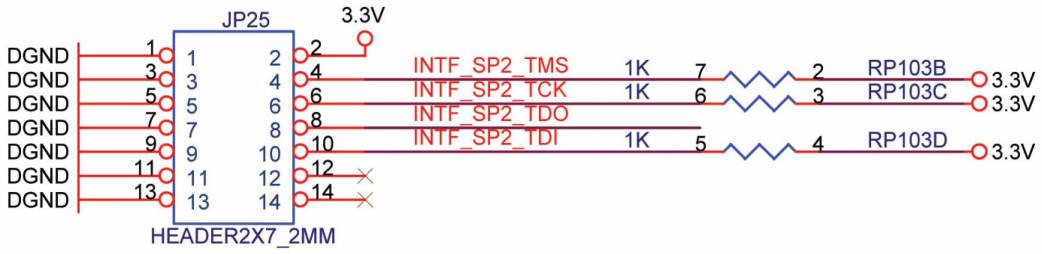
Connector Types:	Header 2x7 2MM
Number of Connections:	14
Mating Connector:	2x7 (14 position) 2MM Connector; Model Part No. 87832-1420

Please visit www.xilinx.com for more information regarding making a cable for Xilinx JTAG or it can be purchased from Xilinx.

The following table gives the pin numbers and functions for the JP25 connector.

Table 24. Xilinx JTAG Connector for XC2S600E-FG456

Pin Number	JP25 Function	Direction (To SBC6713e)
2	3.3V	Power
4	INTF_SP2_TMS	I
6	INTF_SP2_TCK	I
8	INTF_SP2_TDO	O
10	INTF_SP2_TDI	I
1,3,5,7,9,11,13	Digital ground	Power



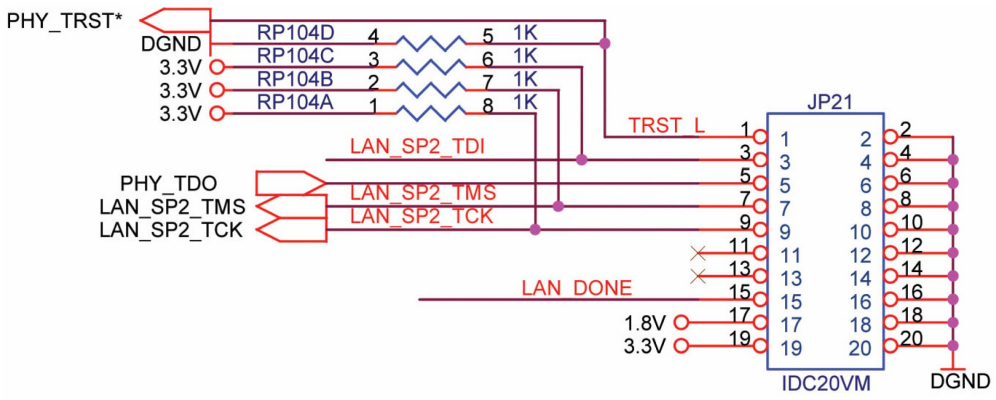
JP21 - Xilinx JTAG Connector (i.e. XC2S50E-TQ144) (Rev C)

Connector Types:	Shrouded header
Number of Connections:	20

The following table gives the pin numbers and functions for the JP25 connector. For more information regarding the cable, please visit www.xilinx.com.

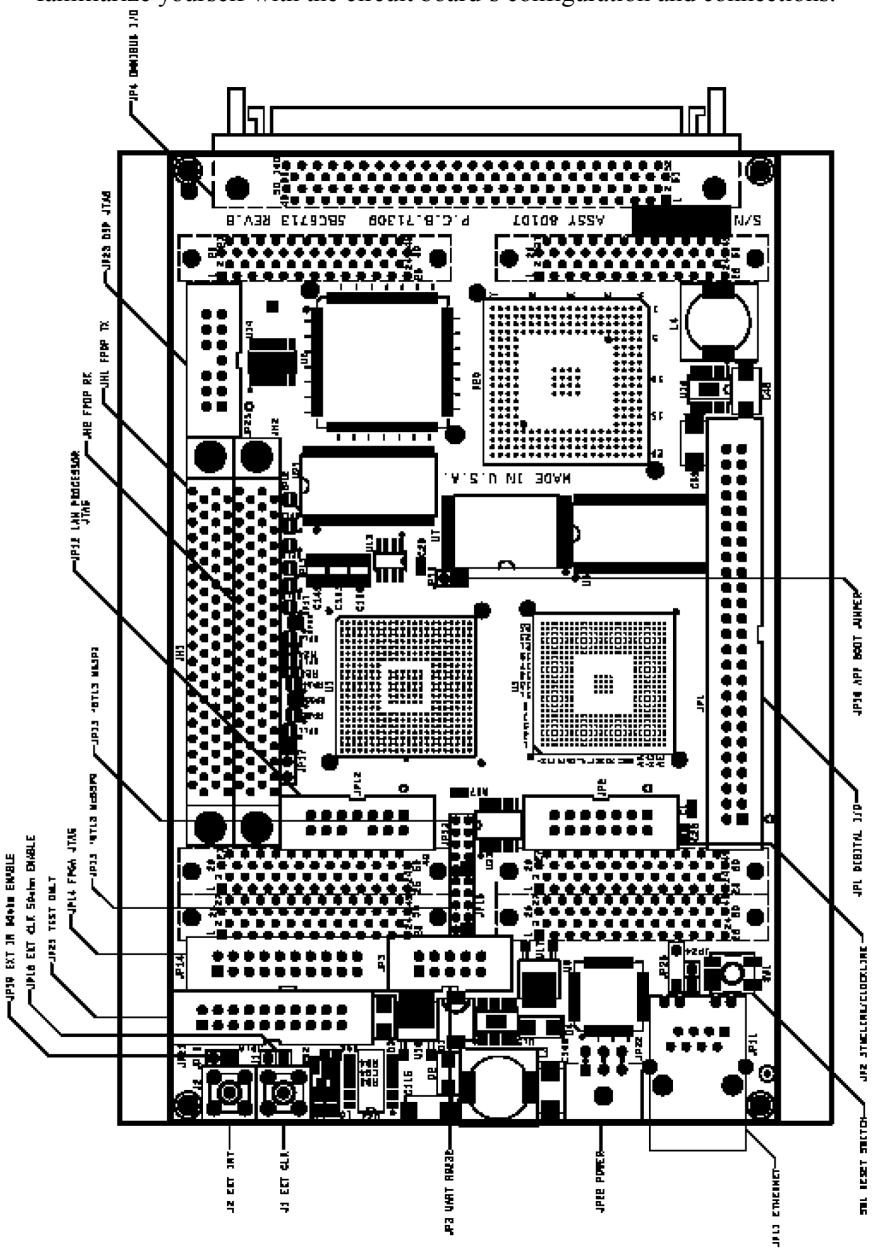
Table 25. Xilinx JTAG Connector for XC2S50E-TQ144.

Pin Number	Function	Direction (from SBC6713e)
1	PHY_TRST	O
3	LAN_SP2_TDI	I
4	PHY_TDO	I
5	LAN_SP2_TMS	O
6	LAN_SP2_TCK	O
15	LAN_DONE	I
17	1.8V	Power
19	3.3V	Power
2,4,6,8,10,12	Digital Ground	Power
14,16,18,20	Digital Ground	Power



Board Layout (Rev B)

A dimensional schematic and the board layout is displayed on the following page. Please review these drawings to familiarize yourself with the circuit board's configuration and connections.



Board Layout (Rev C)

A dimensional schematic and the board layout is displayed on the following page. Please review these drawings to familiarize yourself with the circuit board's configuration and connections.

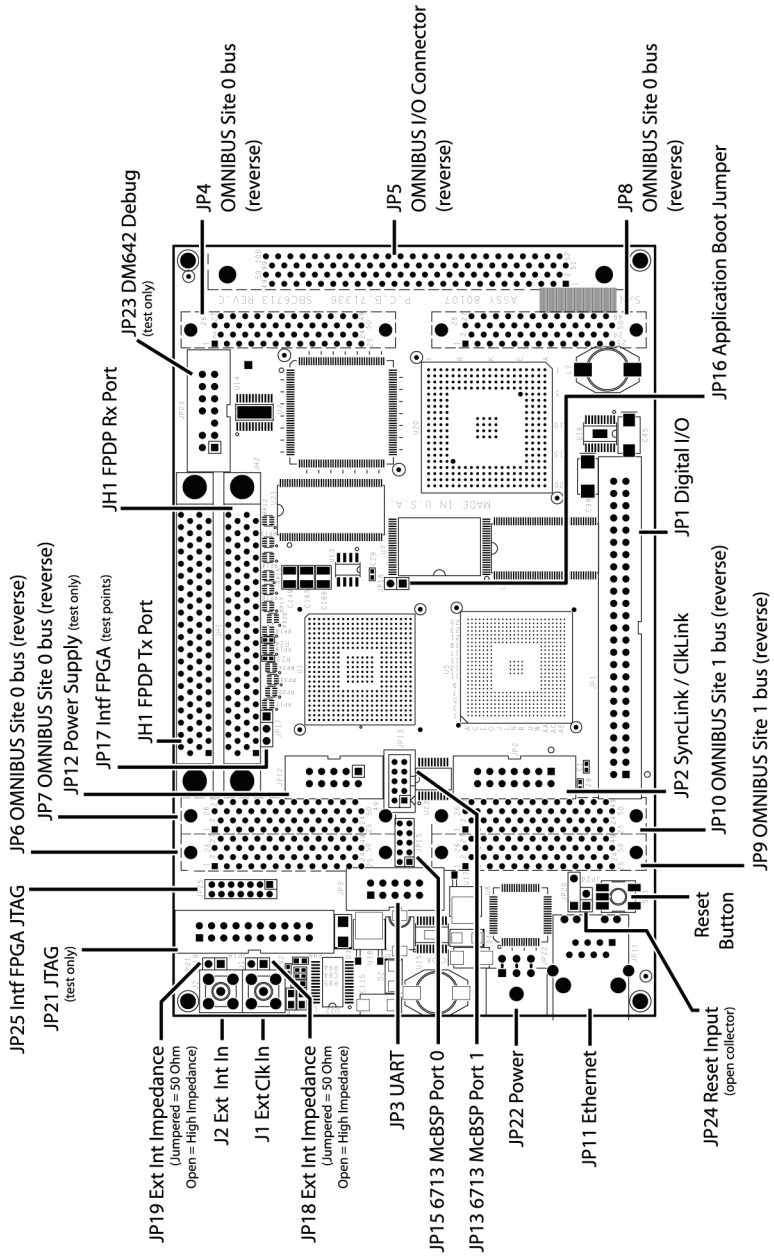
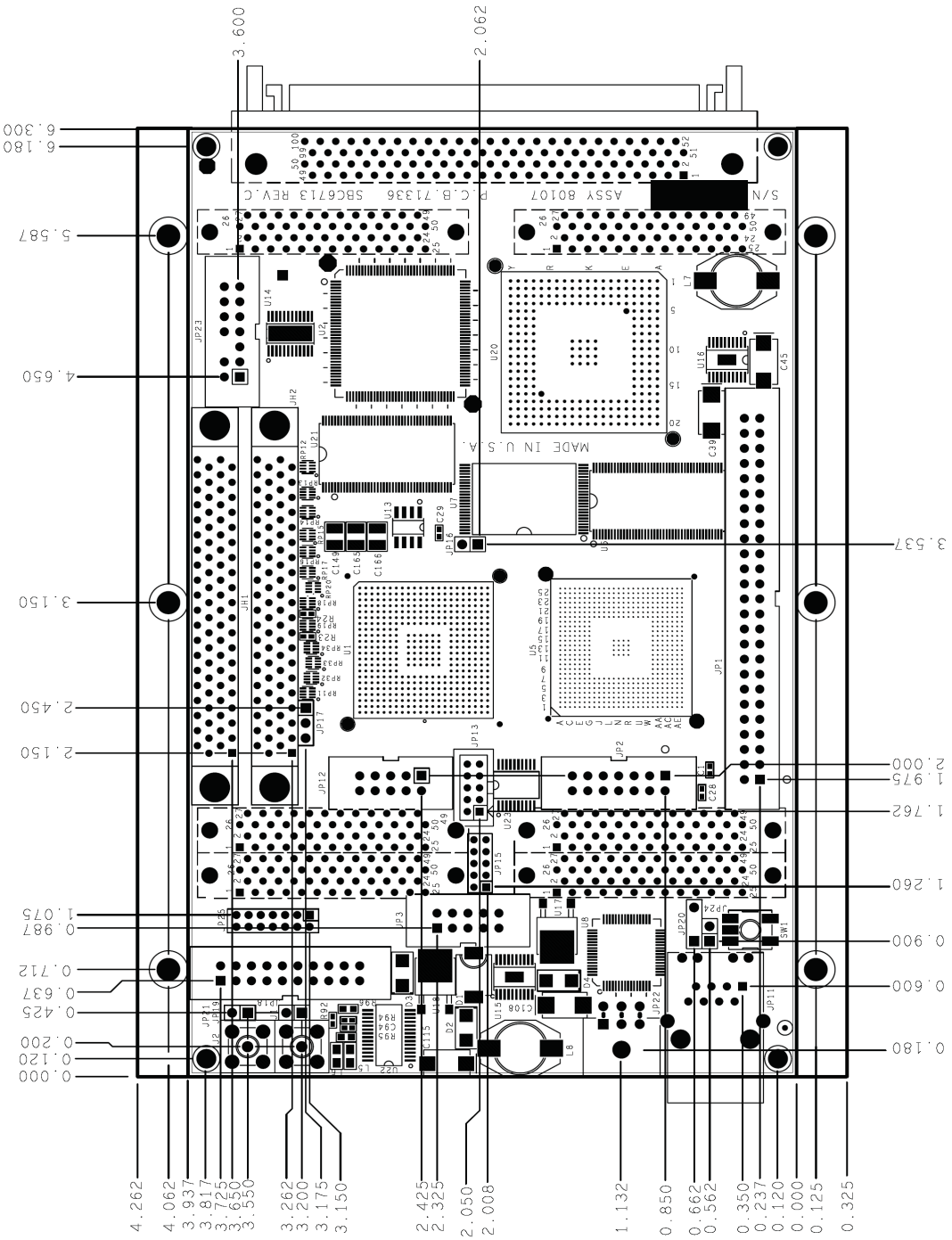

JP4 OMNIBUS Site 0 bus (reverse)
JP5 OMNIBUS I/O Connector (reverse)
JP8 OMNIBUS Site 0 bus (reverse)
JP23 DM642 Debug (test only)
JH1 FPDP Rx Port
JP16 Application Boot Jumper
JP1 Digital I/O
JH1 FPDP Tx Port
JP17 Intf FPGA (test points)
JP12 Power Supply (test only)
JP7 OMNIBUS Site 0 bus (reverse)
JP6 OMNIBUS Site 0 bus (reverse)
JP2 SyncLink / ClkLink
JP10 OMNIBUS Site 1 bus (reverse)
JP9 OMNIBUS Site 1 bus (reverse)
Reset Button
JP25 Intf FPGA JTAG
JP21 JTAG (test only)
JP19 Ext Int Impedance (Jumpered = 50 Ohm Open = High Impedance)
J2 Ext Int In
J1 Ext Clk In
JP18 Ext Int Impedance (Jumpered = 50 Ohm Open = High Impedance)
JP3 UART
JP15 6713 McBSP Port 0
JP13 6713 McBSP Port 1
JP22 Power
JP11 Ethernet
JP24 Reset Input (open collector)

Figure 57. Mechanical Drawing (Board Revision C)


3.600
2.062
6.300
6.180
5.587
4.650
3.537
3.150
2.450
2.150
2.000
1.975
1.762
1.260
0.900
0.600
0.180
1.075
0.987
0.712
0.637
0.425
0.200
0.120
0.000
4.262
4.062
3.937
3.817
3.725
3.650
3.550
3.262
3.200
3.175
3.150
2.425
2.325
2.050
2.008
1.132
0.850
0.662
0.562
0.350
0.237
0.120
0.000
0.125
0.325